A Comprehensive Guide to Trio Technologies on the MX

Juniper MX Series

Douglas Richard Hanks, Jr.
& Harry Reynolds

O'REILLY® JUNIPer NETWORKS

Juniper MX Series

Douglas Richard Hanks, Jr. and Harry Reynolds

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Juniper MX Series
by Douglas Richard Hanks, Jr. and Harry Reynolds

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Mike Loukides and Meghan Blanchette
Development Editor: Patrick Ames
Production Editor: Holly Bauer
Copyeditor: Absolute Service, Inc.
Proofreader: Rachel Leach

Indexer: Bob Pfahler
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Rebecca Demarest

October 2012: First Edition.

Revision History for the First Edition:
2012-09-24 First release
See *http://oreilly.com/catalog/errata.csp?isbn=9781449319717* for release details.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-1-449-31971-7

[LSI]

1348575579

Dedicated to my wife and my parents. You guys are the best. Love you.

—Douglas

I would like to acknowledge my wife, Anita, and our two lovely daughters, Christina and Marissa, for once again understanding and accommodating my desire to engage in this project. And thanks to Doug, that plucky young lad who managed to goad me into engaging in this project when my day job was already rather action-packed. A special thanks to my manager, Andrew Pangelinan at Juniper Networks, for his understanding and support in this project.

—Harry

Table of Contents

About the Authors

Douglas Richard Hanks, Jr. is a Data Center Architect with Juniper Networks and focuses on solution architecture. Previously, he was a Senior Systems Engineer with Juniper Networks, supporting large enterprise accounts such as Chevron, HP, and Zynga. He is certified with Juniper Networks as JNCIE-ENT #213 and JNCIE-SP #875. Douglas' interests are network engineering and architecture for enterprise and service provider technologies. He is the author of several *Day One* books published by Juniper Networks Books. Douglas is also the cofounder of the Bay Area Juniper Users Group (BAJUG). When he isn't busy with networking, Douglas enjoys computer programming, photography, and Arduino hacking. Douglas can be reached at *doug@juniper.net* or on Twitter *@douglashanksjr*.

Harry Reynolds has over 30 years of experience in the networking industry, with the last 20 years focused on LANs and LAN interconnection. He is CCIE # 4977 and JNCIE # 3 and also holds various other industry and teaching certifications. Harry was a contributing author to *Juniper Network Complete Reference* (McGraw-Hill) and wrote the *JNCIE and JNCIP Study Guides* (Sybex Books). He coauthored *Junos Enterprise Routing* and *Junos Enterprise Switching* (O'Reilly). Prior to joining Juniper, Harry served in the US Navy as an Avionics Technician, worked for equipment manufacturer Micom Systems, and spent much time developing and presenting hands-on technical training curricula targeted to both enterprise and service provider needs. Harry has developed and presented internetworking classes for organizations such as American Institute, American Research Group, Hill Associates, and Data Training Resources. Currently, Harry performs Customer Specific Testing that simulates one of the nation's largest private IP backbones at multidimensional scale. When the testing and writing is done (a rare event, to be sure), Harry can be found in his backyard metal shop trying to make Japanese-style blades.

About the Lead Technical Reviewers

Stefan Fouant is a Technical Trainer and JNCP Proctor at Juniper Networks with over 15 years of experience in the networking industry. His first exposure to Junos was with Junos 3.4 on the original M40 back in 1998, and it has been a love affair ever since. His background includes launching an industry-first DDoS Mitigation and Detection service at Verizon Business, as well as building customized solutions for various mission-critical networks. He holds several patents in the areas of DDoS Detection and Mitigation, as well as many industry certifications including CISSP, JNCIE-SP, JNCIE-ENT, and JNCIE-SEC.

Artur Makutunowicz has over five years of experience in Information Technology. He was a Technical Team Leader at a large Juniper Elite partner. His main areas of interest are Service Provider technologies, network device architecture, and Software Defined Networking (SDN). He was awarded with JNCIE-ENT #297 certification. Artur was also a technical reviewer of *Day One: Scaling Beyond a Single Juniper SRX in the Data Center*, published by Juniper Networks Books. He is currently an independent contractor and can be reached at *artur@makutunowicz.net*.

About the Technical Reviewers

Many Junos engineers reviewed this book. They are, in the authors' opinion, some of smartest and most capable networking people around. They include but are not limited to: Kannan Kothandaraman, Ramesh Prabagaran, Dogu Narin, Russell Gerald Kelly, Rohit Puri, Sunesh Rustagi, Ajay Gaonkar, Shiva Shenoy, Massimo Magnani, Eswaran Srinivasan, Nitin Kumar, Ariful Huq, Nayan Patel, Deepak Ojha, Ramasamy Ramanathan, Brandon Bennett, Scott Mackie, Sergio Danelli, Qi-Zhong Cao, Eric Cheung Young Sen, Richard Fairclough, Madhu Kopalle, Jarek Sawczuk, Philip Seavey, and Amy Buchanan.

Proof of Concept Laboratory

In addition, the authors humbly thank the POC Lab in Sunnyvale, California, where the test bed for this book was cared for and fed by Roberto Hernandez, Ridha Hamidi, and Matt Bianchi. Without access to test equipment, this book would have been impossible.

Preface

One of the most popular routers in the enterprise and service provider market is the Juniper MX Series. The industry is moving to high-speed, high port-density Ethernet-based routers, and the Juniper MX was designed from the ground up to solve these challenges.

This book is going to show you, step-by-step, how to build a better network using the Juniper MX—it's such a versatile platform that it can be placed in the core, aggregation, or edge of any type of network and provide instant value. The Juniper MX was designed to be a network virtualization beast. You can virtualize the physical interfaces, logical interfaces, control plane, data plane, network services, and even have virtualized services span several Juniper MX routers. What was traditionally done with an entire army of routers can now be consolidated and virtualized into a single Juniper MX router.

No Apologies

We're avid readers of technology books, and we always get a bit giddy when a new book is released because we can't wait to read it and learn more about a specific technology. However, one trend we have noticed is that every networking book tends to regurgitate the basics over and over. There are only so many times you can force yourself to read about spanning tree, the split horizon rule, or OSPF LSA types. One of the goals of this book is to introduce new and fresh content that hasn't been published before.

There was a conscious decision made between the authors to keep the technical quality of this book very high; this created a constant debate whether or not to include primer or introductory material in the book to help refresh a reader's memory with certain technologies and networking features. In short, here's what we decided:

Spanning Tree

There's a large chapter on bridging, VLAN mapping, IRB, and virtual switches. A logical choice would be to include the spanning tree protocol in this chapter. However, spanning tree has been around forever and quite frankly there's nothing special or interesting about it. Spanning tree is covered in great detail in every JNCIA and CCNA book on the market. If you want to learn more about spanning tree check out *Junos Enterprise Switching* by O'Reilly or *CCNA ICND2 Official Exam and Certification Guide*, Second Edition by Cisco Press.

Basic Firewall Filters
We decided to skip the basic firewall filter introduction and jump right into the advanced filtering and policing that's available on the Juniper MX. Hierarchical policers, two-rate three-color policers, and cascading firewall filters are much more interesting.

Class of Service
This was a difficult decision because Chapter 5 is over 170 pages of advanced hierarchal class of service. Adding another 50 pages of class of service basics would have exceeded page count constraints and provided no additional value. If you would like to learn more about basic class of service check out *QoS-Enabled Networks* by Wiley, *Junos Enterprise Routing*, Second Edition by O'Reilly, or *Juniper Networks Certified Internet Expert Study Guide* by Juniper Networks.

Routing Protocols
There are various routing protocols such as OSPF and IS-IS used throughout this book in case studies. No introduction chapters are included for IS-IS or OSPF, and it's assumed that you are already familiar with these routing protocols. If you want to learn more about OSPF or IS-IS, check out the *Junos Enterprise Routing*, Second Edition by O'Reilly or *Juniper Networks Certified Internet Expert Study Guide* by Juniper Networks.

Virtual Chassis
This was an interesting problem to solve. On one hand, virtual chassis was covered indepth in the book *Junos Enterprise Switching* by O'Reilly, but on the other hand there are many caveats and features that are only available on the Juniper MX. It was decided to provide enough content in the introduction that a new user could grasp the concepts, but someone already familiar with virtual chassis wouldn't become frustrated. Chapter 6 specifically focuses on the technical prowess of virtual chassis and the Juniper MX implementation of virtual chassis.

After many hours of debate over Skype, it was decided that we should defer to other books when it comes to introductory material and keep the content of this book at an expert level. We expect that most of our readers already have their JNCIE or CCIE (or are well on their way) and will enjoy the technical quality of this book. For beginning readers, we want to share an existing list of books that are widely respected within the networking community:

Junos Enterprise Routing, Second Edition, O'Reilly
Junos Enterprise Switching, O'Reilly
Junos Cookbook, O'Reilly
Junos Security, O'Reilly
Junos High Availability, O'Reilly
QoS-Enabled Networks, Wiley & Sons

MPLS-Enabled Applications, Third Edition, Wiley & Sons
Network Mergers and Migrations, Wiley
Juniper Networks Certified Internet Expert, Juniper Networks
Juniper Networks Certified Internet Professional, Juniper Networks
Juniper Networks Certified Internet Specialist, Juniper Networks
Juniper Networks Certified Internet Associate, Juniper Networks
CCIE Routing and Switching, Fourth Edition, Cisco Press
Routing TCP/IP, Volumes I and II, Cisco Press
OSPF and IS-IS, Addison-Wesley
OSPF: Anatomy of an Internet Routing Protocol, Addison-Wesley
The Art of Computer Programming, Addison-Wesley
TCP/*IP Illustrated*, Volumes 1, 2, and 3, Addison-Wesley
UNIX Network Programming, Volumes 1 and 2, Prentice Hall PTR
Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices, Morgan Kaufmann

Book Topology

Using the same methodology found in the JNCIP-M and JNCIE-M Study Guides, this book will use a master topology and each chapter will use a subset of the devices that are needed to illustrate features and case studies. The master topology is quite extensive and includes four Juniper MX240s, two EX4500s, two EX4200s, and various port testers which can generate traffic and emulate peering and transit links. The topology is broken into three major pieces:

Data Center 1
The left side of the topology represents Data Center 1. The devices include W1, W2, S1, S2, R1, R2, P1, and T2. The address space can be summarized as 10.0.0.0/14.

Data Center 2
The right side of the topology represents Data Center 2. It's common for networks to have more than one data center, so it made sense to create a master topology that closely resembles a real production network. The devices include W3, W4, S3, S4, R3, R4, P2, and T2.

The Core
The core is really just a subset of the two data centers combined. Typically when interconnecting data centers a full mesh of WAN links aren't cost effective, so we decided to only use a pair of links between Data Center 1 and Data Center 2.

For the sake of clarity and readability, the master topology has been broken into five figures, Figure P-1 through Figure P-5: Interface Names, Aggregate Ethernet Assignments, Layer 2, IPv4 Addressing, and IPv6 Addressing. The breakdown and configuration of the equipment is as follows:

W1: *Web Server 1. This is a tester port that's able to generate traffic.*
W2: *Web Server 2. This is a tester port that's able to generate traffic.*
S1: *Access Switch 1. This is a Juniper EX4500 providing both Layer 2 and Layer 3 access.*
S2: *Access Switch 2. This is a Juniper EX4500 providing both Layer 2 and Layer 3 access.*
R1: *Core Router/WAN Router 1. Juniper MX240 with a MPC2 Enhanced Queuing line card.*
R2: *Core Router/WAN Router 2. Juniper MX240 with a MPC2 Enhanced Queuing line card.*
R3: *Core Router/WAN Router 3. Juniper MX240 with a MPC2 line card.*
R4: *Core Router/WAN Router 4. Juniper MX240 with a MPC2 Queuing line card.*
S3: *Access Switch 3. Juniper EX4200 providing both Layer 2 and Layer 3 access.*
S4: *Access Switch 4. Juniper EX4200 providing both Layer 2 and Layer 3 access.*
W3: *Web Server 3. This is a tester port that's able to generate traffic.*
W4: *Web Server 4. This is a tester port that's able to generate traffic.*
P1: *Peering Router 1. This is a tester port that's able to generate traffic.*
P2: *Peering Router 2. This is a tester port that's able to generate traffic.*
T1: *Transit Router 1. This is a tester port that's able to generate traffic.*
T2: *Transit Router 2. This is a tester port that's able to generate traffic.*

Interface Names

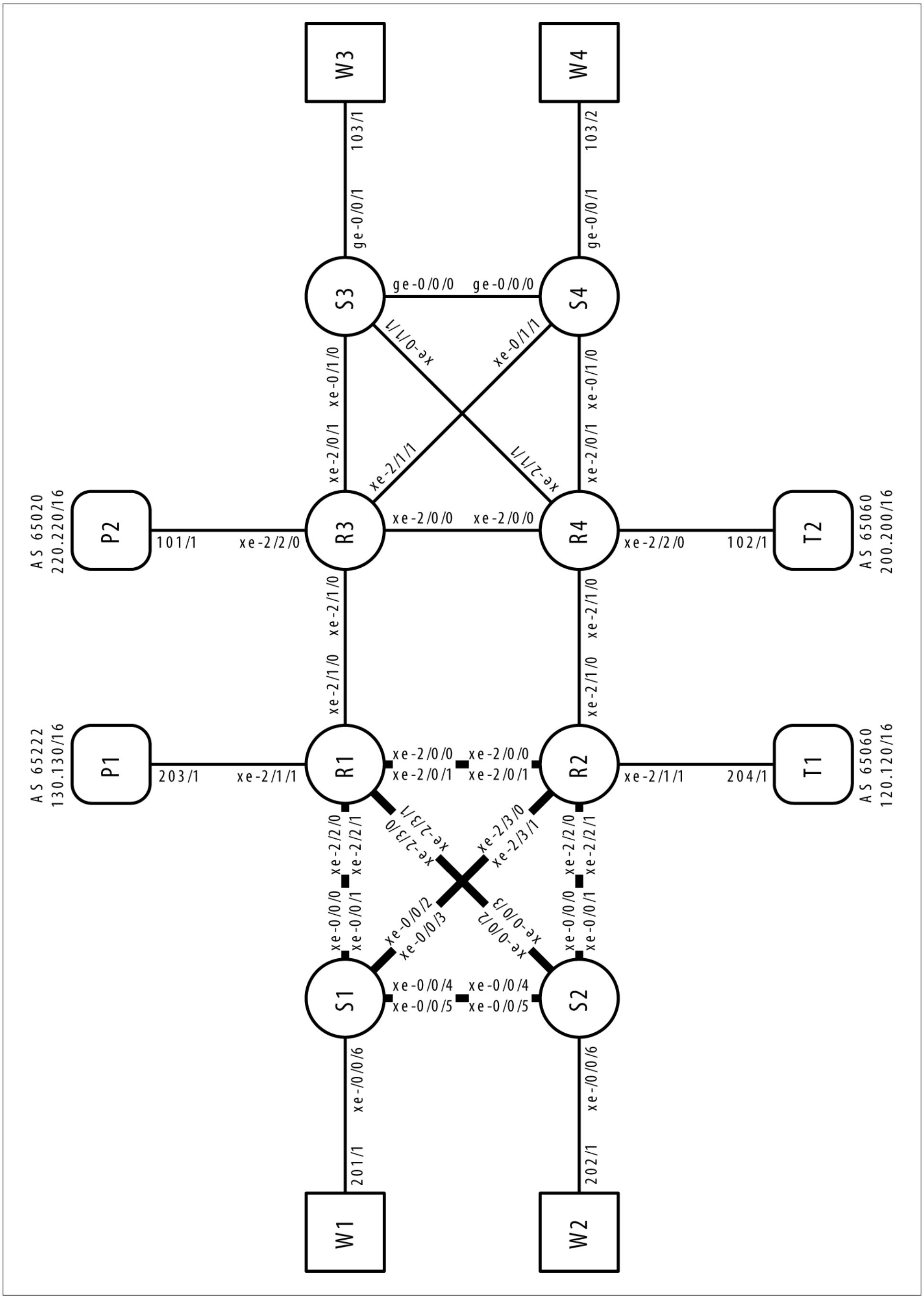


Figure P-1. Master topology: Interface names

Aggregate Ethernet Assignments

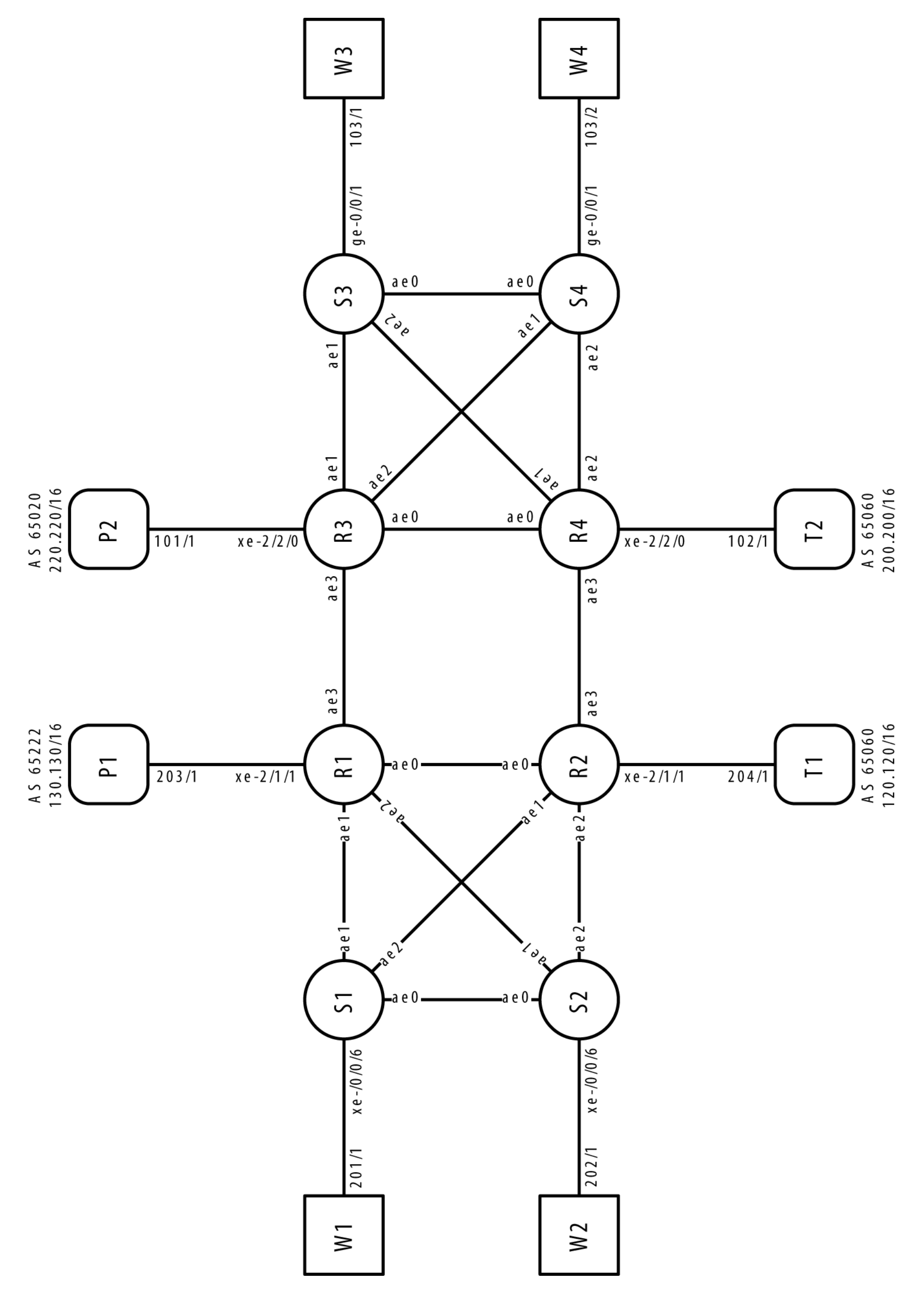


Figure P-2. Master topology: Aggregate ethernet assignments

Layer 2

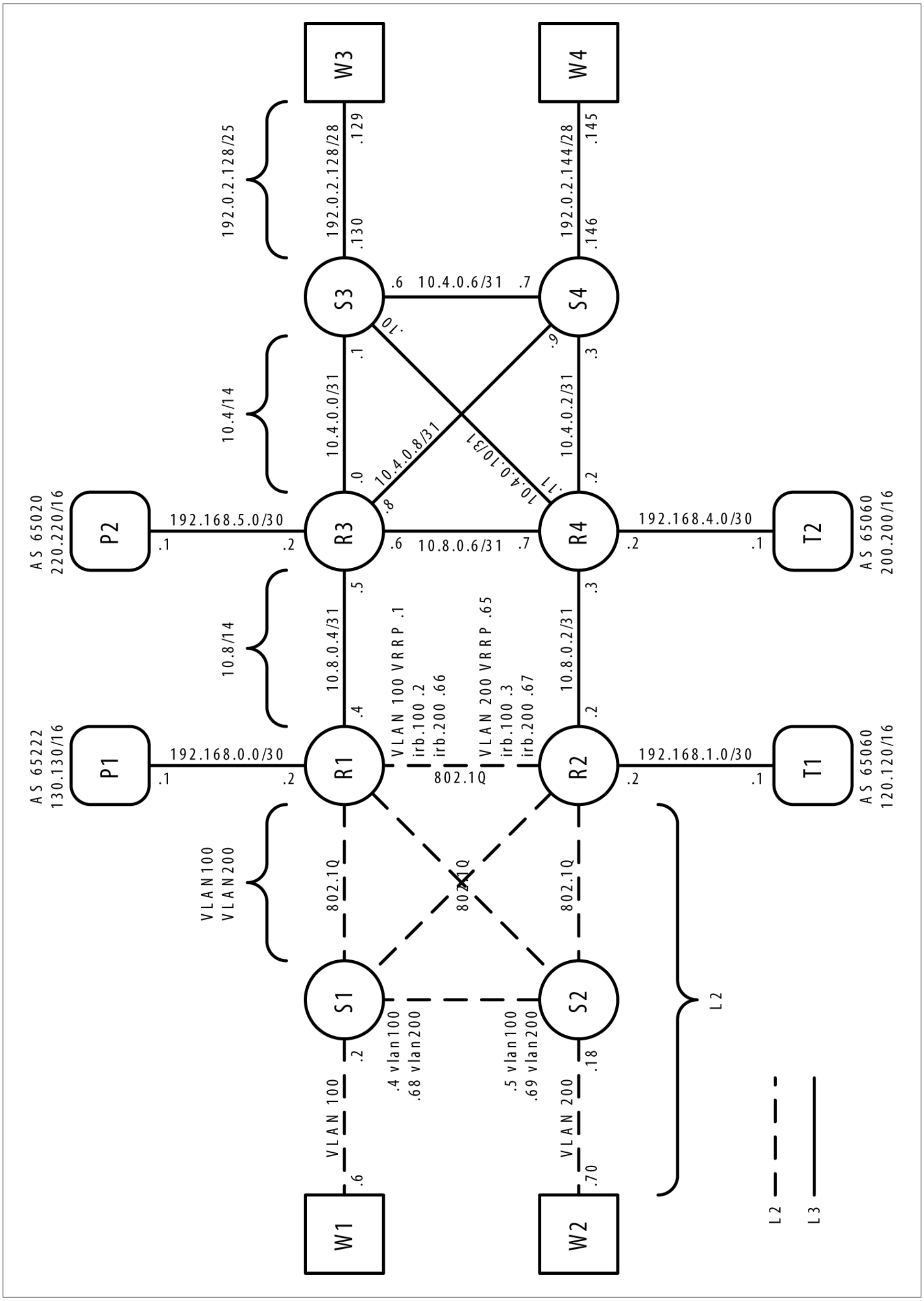


Figure P-3. Master topology: Layer 2

IPv4 Addressing

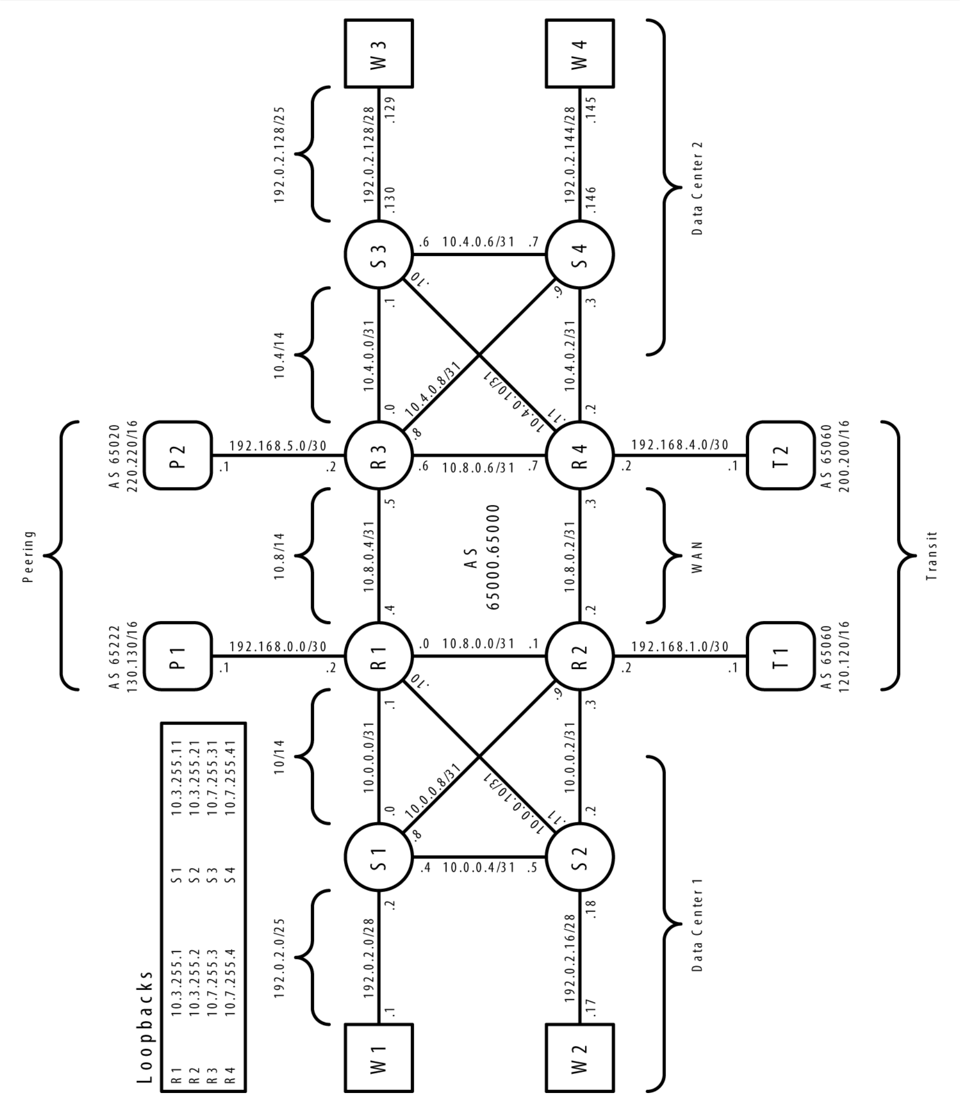
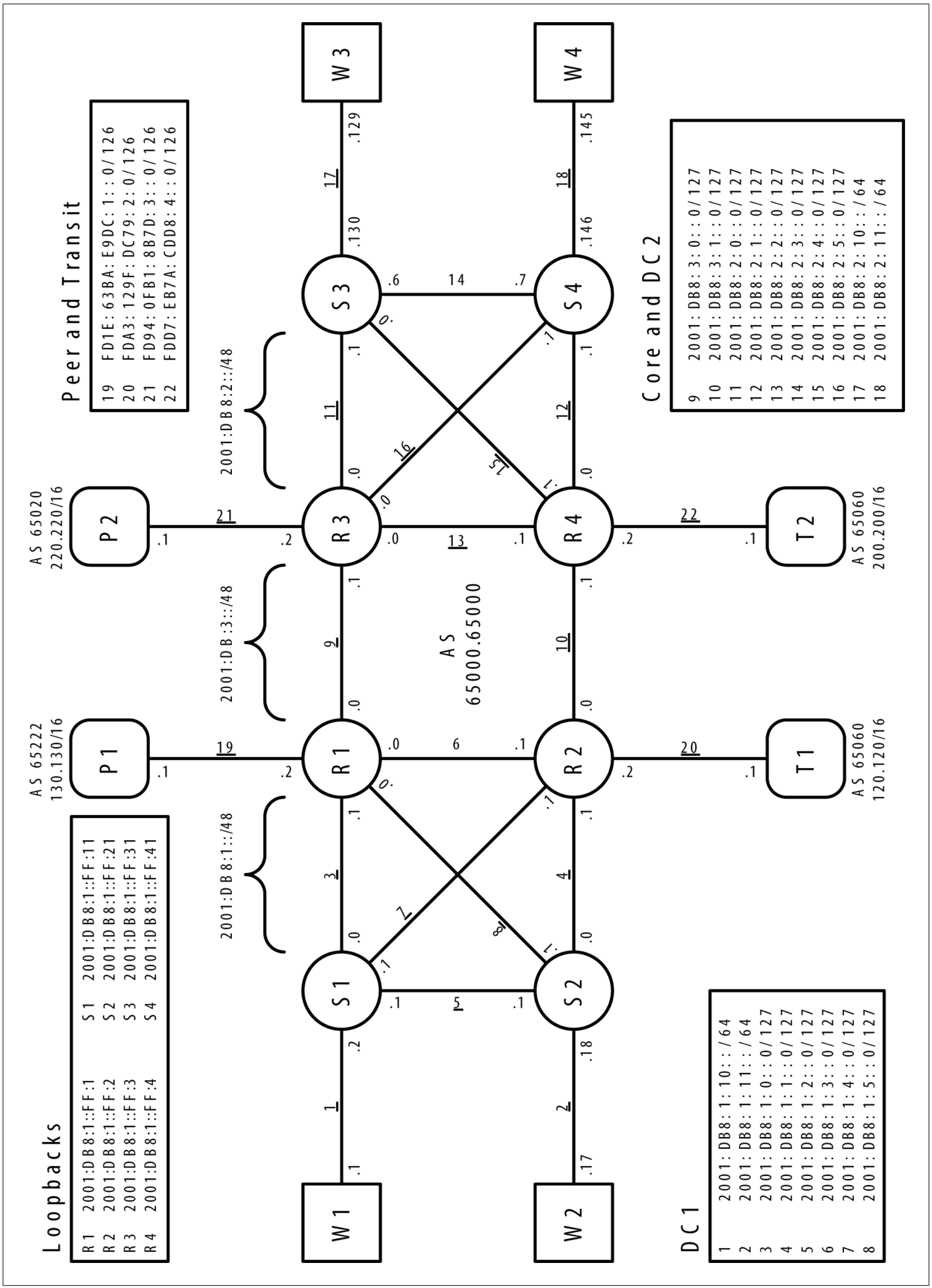


Figure P-4. Master topology: IPv4 addressing

IPv6 Addressing



Loopbacks

R1	2001:DB8:1::FF:1	S1	2001:DB8:1::FF:11
R2	2001:DB8:1::FF:2	S2	2001:DB8:1::FF:21
R3	2001:DB8:1::FF:3	S3	2001:DB8:1::FF:31
R4	2001:DB8:1::FF:4	S4	2001:DB8:1::FF:41

DC1

1	2001:DB8:1:10::/64
2	2001:DB8:1:11::/64
3	2001:DB8:1:0::0/127
4	2001:DB8:1:1::0/127
5	2001:DB8:1:2::0/127
6	2001:DB8:1:3::0/127
7	2001:DB8:1:4::0/127
8	2001:DB8:1:5::0/127

Core and DC2

9	2001:DB8:3:0::0/127
10	2001:DB8:3:1::0/127
11	2001:DB8:2:0::0/127
12	2001:DB8:2:1::0/127
13	2001:DB8:2:2::0/127
14	2001:DB8:2:3::0/127
15	2001:DB8:2:4::0/127
16	2001:DB8:2:5::0/127
17	2001:DB8:2:10::/64
18	2001:DB8:2:11::/64

Peer and Transit

19	FD1E:63BA:E9DC:1::0/126
20	FDA3:129F:DC79:2::0/126
21	FD94:0FB1:8B7D:3::0/126
22	FDD7:EB7A:CDD8:4::0/126

Figure P-5. Master topology: IPv6 addressing

What's in This Book?

This book was written for network engineers by network engineers. The ultimate goal of this book is to share with the reader the logical underpinnings of the Juniper MX. Each chapter represents a specific vertical within the Juniper MX and will provide enough depth and knowledge to provide the reader with enough confidence to implement and design new architectures for their network using the Juniper MX.

Here's a short summary of the chapters and what you'll find inside:

Chapter 1, Juniper MX Architecture
: Learn a little bit about the history and pedigree of the Juniper MX and what factors prompted its creation. Junos is the "secret sauce" that's common throughout all of the hardware; this chapter will take a deep dive into the control plane and explain some of the recent important changes to the release cycle and support structure of Junos. The star of the chapter is of course the Juniper MX; the chapter will thoroughly explain all of the components such as line cards, switch fabric, and routing engines.

Chapter 2, Bridging, VLAN Mapping, IRB, and Virtual Switches
: It always seems to surprise people that the Juniper MX is capable of switching; not only can it switch, it has some of the best bridging features and scalability on the market. The VLAN mapping is capable of popping, swapping, and pushing new IEEE 802.1Q headers with ease. When it comes to scale, it can support over 8,000 virtual switches.

Chapter 3, Stateless Filters, Hierarchical Policing, and Tri-Color Marking
: Discover the world of advanced policing where the norm is creating two-rate three-color markers, hierarchical policers, cascading firewall filters, and logical bandwidth policers. You think you already know about Junos policing and firewall filters? You're wrong; this is a must-read chapter.

Chapter 4, Routing Engine Protection and DDoS Prevention
: Everyone has been through the process of creating a 200-line firewall filter and applying it to the loopback interface to protect the routing engine. This chapter presents an alternative method of creating a firewall filter framework and only applies the filters that are specific to your network via firewall filter chains. As of Junos 10.4, there's a new feature called Distributed Denial-of-Service Protection (`ddos-protection`) that can be combined with firewall filters to add an extra layer of security to the routing engine.

Chapter 5, Trio Class of Service
: This chapter answers the question, "What is hierarchical class of service and why do I need it?" The land of CoS is filled with mystery and adventure; come join Harry and discover the advantages of hierarchical scheduling.

Chapter 6, MX Virtual Chassis
: What's better than a Juniper MX router? Two Juniper MX routers, of course, unless you're talking about virtual chassis; it takes several Juniper MX Routers and combines them into a single, logical router.

Chapter 7, Trio Inline Services
: Services such as Network Address Translation (NAT), IP Information Flow Export (IPFIX), and tunneling protocols traditionally require a separate services card. Trio inline services turns this model upside down and allows the network engineer to install network services directly inside of the Trio chipset, which eliminates the need for special services hardware.

Chapter 8, Multi-Chassis Link Aggregation
: An alternative to virtual chassis is a feature called MC-LAG, which allows two routers to form a logical IEEE 802.3ad connection to a downstream router and appear as a single entity. The twist is that MC-LAG allows the two routers to function independently.

Chapter 9, Junos High Availability on MX Routers
: Some of us take high availability for granted. GRES, NSR, NSB, and ISSU make you feel warm and fuzzy. But how do you really know they work? Put on your hard hat and go spelunking inside of these features and protocols like you never have before.

Each chapter includes a set of review questions and exam topics, all designed to get you thinking about what you've just read and digested. If you're not in the certification mode, the questions will provide a mechanism for critical thinking, potentially prompting you to locate other resources to further your knowledge.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, file extensions, pathnames, directories, and Unix utilities.

`Constant width`
: Indicates commands, options, switches, variables, attributes, keys, functions, types, classes, namespaces, methods, modules, properties, parameters, values, objects, events, event handlers, XML tags, macros, the contents of files, and the output from commands.

`Constant width bold`
: Shows commands and other text that should be typed literally by the user, as well as important lines of code.

`Constant width italic`

Shows text that should be replaced with user-supplied values.

This icon signifies a tip, suggestion, or general note.

This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your own configuration and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the material. For example, deploying a network based on actual configurations from this book does not require permission. Selling or distributing a CD-ROM of examples from this book does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of sample configurations or operational output from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN, for example: "*Juniper MX Series* by Douglas Richard Hanks, Jr., and Harry Reynolds. Copyright 2012, Douglas Hanks, Jr., and Harry Reynolds, 978-1-449-31971-7."

If you feel your use of code examples falls outside fair use or the permission given here, feel free to contact us at *permissions@oreilly.com*.

As with most deep-dive books, the reader will be exposed to a variety of hidden, Junos Shell, and even MPC-level VTY commands performed after forming an internal connection to a PFE component. As always, the standard disclaimers apply.

In general, a command being hidden indicates that the feature is not officially supported in that release. Such commands should only be used in production networks after consultation with JTAC. Likewise, the shell is not officially supported or documented. The commands available can change, and you can render a router unbootable with careless use of shell commands. The same holds true for PFE component-level shell commands, often called VTY commands, that, again, when undocumented, are capable of causing network disruption or damage to the routing platform that can render it inoperable.

The hidden and shell commands that are used in this book were selected because they were the only way to illustrate certain operational characteristics or the results of complex configuration parameters.

Again, hidden and shell commands should only be used under JTAC guidance; this is especially true when dealing with a router that is part of a production network.

You have been duly warned.

Safari® Books Online

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://oreil.ly/juniper_mx_series* or *http://cubednetworks.com*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

CHAPTER 1
Juniper MX Architecture

Back in 1998, Juniper Networks released its first router, the M40. Leveraging Application Specific Integrated Circuits (ASICs), the M40 was able to outperform any other router architecture. The M40 was also the first router to have a true separation of the control and data planes, and the M Series was born. Originally, the model name M40 referred to its ability to process 40 million packets per second (Mpps). As the product portfolio expanded, the “M” now refers to the multiple services available on the router, such as MPLS with a wide variety of VPNs. The primary use case for the M Series was to allow Service Providers to deliver services based on IP while at the same time supporting legacy frame relay and ATM networks.

Fast forward 10 years and the number of customers that Service Providers have to support has increased exponentially. Frame relay and ATM have been decimated, as customers are demanding high-speed Layer 2 and Layer 3 Ethernet-based services. Large enterprise companies are becoming more Service Provider-like and are offering IP services to departments and subsidiaries.

Nearly all networking equipment connects via Ethernet. It’s one of the most well understood and deployed networking technologies used today. Companies have challenging requirements to reduce operating costs and at the same time provide more services. Ethernet enables the simplification in network operations, administration, and maintenance.

The MX Series was introduced in 2007 to solve these new challenges. It is optimized for delivering high-density and high-speed Layer 2 and Layer 3 Ethernet services. The “M” still refers to the multiple services heritage, while the “X” refers to the new switching capability and focus on 10G interfaces and beyond; it’s also interesting to note that the Roman numeral for the number 10 is “X.”

It’s no easy task to create a platform that’s able to solve these new challenges. The MX Series has a strong pedigree: although mechanically different, it leverages technology from both the M and T Series for chassis management, switching fabric, and the routing engine.

Features that you have come to know and love on the M and T Series are certainly present on the MX Series as it runs on the same image of Junos. In addition to the "oldies, but goodies," is an entire featureset focused on Service Provider switching and broadband network gateway (BNG). Here's just a sample of what is available on the MX:

High availability
: Non-Stop Routing (NSR), Non-Stop Bridging (NSB), Graceful Routing Engine Switch over (GRES), Graceful Restart (GR), and In-Service Software Upgrade (ISSU)

Routing
: RIP, OSPF, IS-IS, BGP, and Multicast

Switching
: Full suite of Spanning Tree Protocols (STP), Service Provider VLAN tag manipulation, QinQ, and the ability to scale beyond 4,094 bridge domains by leveraging virtual switches

Inline services
: Network Address Translation (NAT), IP Flow Information Export (IPFIX), Tunnel Services, and Port Mirroring

MPLS
: L3VPN, L2VPNs, and VPLS

Broadband services
: PPPoX, DHCP, Hierarchical QoS, and IP address tracking

Virtualization
: Multi-Chassis Link Aggregation, Virtual Chassis, Logical Systems, Virtual Switches

With such a large featureset, the use case of the MX Series is very broad. It's common to see it in the core of a Service Provider network, providing BNG, or in the Enterprise providing edge routing or core switching.

This chapter introduces the MX platform, features, and architecture. We'll review the hardware, components, and redundancy in detail.

Junos

Junos is a purpose-built networking operating system based on one of the most stable and secure operating systems in the world: FreeBSD. Junos is designed as a monolithic kernel architecture that places all of the operating system services in the kernel space. Major components of Junos are written as daemons that provide complete process and memory separation.

One of the benefits of monolithic kernel architecture is that kernel functions are executed in supervisor mode on the CPU while the applications and daemons are executed in user space. A single failing daemon will not crash the operating system or impact other unrelated daemons. For example, if there was an issue with the SNMP daemon and it crashed, it wouldn't impact the routing daemon that handles OSPF and BGP.

One Junos

Creating a single network operating system that's able to be leveraged across routers, switches, and firewalls simplifies network operations, administration, and maintenance. Network operators need only learn Junos once and become instantly effective across other Juniper products. An added benefit of a single Junos is that there's no need to reinvent the wheel and have 10 different implementations of BGP or OSPF. Being able to write these core protocols once and then reuse them across all products provides a high level of stability, as the code is very mature and field tested.

Software Releases

Every quarter for nearly 15 years there has been a consistent and predictable release of Junos. The development of the core operating system is a single release train. This allows developers to create new features or fix bugs once and share them across multiple platforms.

The release numbers are in a major and minor format. The major number is the version of Junos for a particular calendar year and the minor release indicates which quarter the software was released. This happens to line up nicely for Junos 11 and Junos 12 as they directly tied the year released. For example, Junos 11 was released in 2011.

This wasn't always the case. Before Junos 10.1, the major release didn't line up to the year released. Historically, the ".0" release was reserved for major events such as releasing software for new products like the MX240 with Junos 9.0.

Each release of Junos is supported for 18 months. The last release of Junos in the calendar year is known as the Extended End of Life (EEOL), and this release is supported for 36 months.

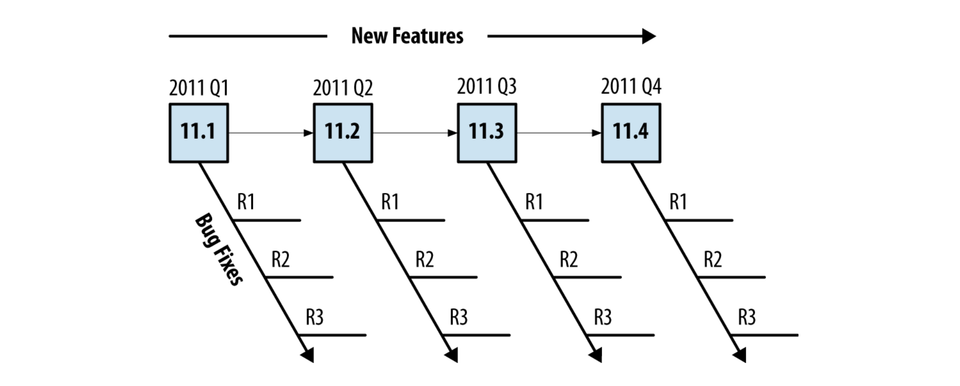


Figure 1-1. Junos release model

There are a couple of different types of Junos that are released more frequently to resolve issues: maintenance and service releases. Maintenance releases are released about every six weeks to fix a collection of issues and they are prefixed with "R." For example, Junos 11.1R2 would be the second maintenance release for Junos 11.1. Service releases are released on demand to specifically fix a critical issue that has yet to be addressed by a maintenance release. These releases are prefixed with a "S." An example would be Junos 11.1S2.

The general rule of thumb is that new features are added every minor release and bug fixes are added every maintenance release. For example, Junos 11.1 to 11.2 would introduce new features, whereas Junos 11.1R1 to 11.1R2 would introduce bug fixes.

Most production networks prefer to use the last Junos release of the previous calendar year; these Junos releases are EEOL releases that are supported for three years. The advantage is that the EEOL releases become more stable with time. Consider that 11.1 will stop providing bug fixes after 18 months, while 11.4 will continue to have bug fixes for 36 months.

Three Release Cadence

In 2012, Junos created a new release model to move from four releases per year to three. This increased the frequency of maintenance releases to resolve more issues more often. The other benefit is that all Junos releases as of 2012 are supported for 24 months, while the last release of Junos for the calendar year will still be considered EEOL and have support for 36 months.

Table 1-1. Junos End of Engineering and End-of-Life schedule

Release	Target	End of Engineering	End of Life
Junos 12.1	March	24 months	+ 6 months
Junos 12.2	July	24 months	+ 6 months

Release	Target	End of Engineering	End of Life
Junos 12.3	November	36 months	+ 6 months

By extending the engineering support and reducing the number of releases, network operators should be able to reduce the frequency of having to upgrade to a new release of code.

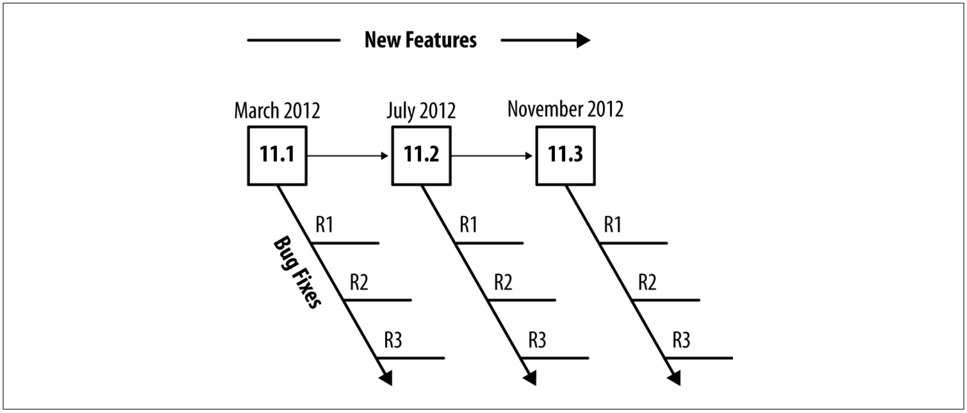


Figure 1-2. New 2012 Junos three-release candidate

With the new Junos three-release cadence, network operators can feel more confident using any version of Junos without feeling pressured to only use the EEOL release.

Software Architecture

Junos was designed from the beginning to support a separation of control and forwarding plane. This is true for the MX Series, where all of the control plane functions are performed by the routing engine while all of the forwarding is performed by the packet forwarding engine (PFE). Providing this level of separation ensures that one plane doesn't impact the other. For example, the forwarding plane could be routing traffic at line-rate and performing many different services while the routing engine sits idle and unaffected.

Control plane functions come in many shapes and sizes. There's a common misconception that the control plane only handles routing protocol updates. In fact, there are many more control plane functions. Some examples include:

- Updating the routing table
- Answering SNMP queries
- Processing SSH or HTTP traffic to administer the router
- Changing fan speed

- Controlling the craft interface
- Providing a Junos micro kernel to the PFEs
- Updating the forwarding table on the PFEs

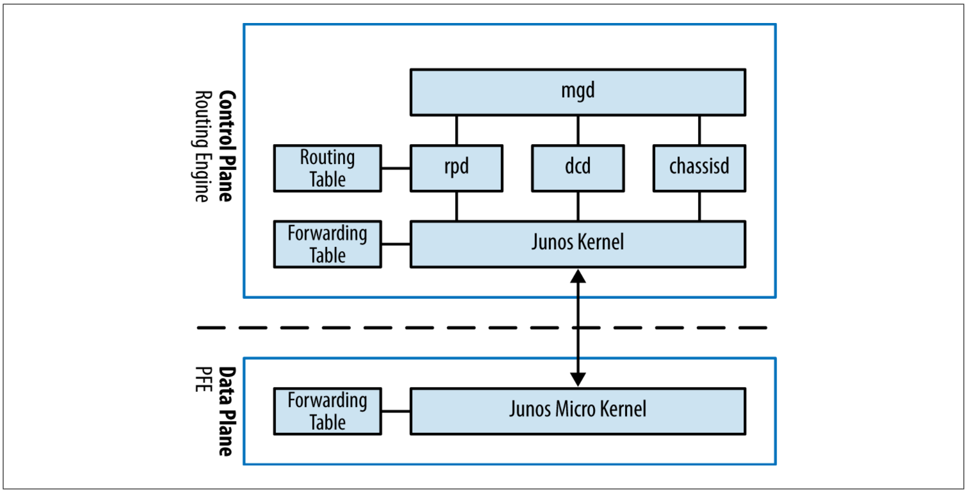


Figure 1-3. Junos software architecture

At a high level, the control plane is implemented entirely within the routing engine while the forwarding plane is implemented within each PFE using a small, purpose-built kernel that contains only the required functions to route and switch traffic.

The benefit of control and forwarding separation is that any traffic that is being routed or switched through the router will always be processed at line-rate on the PFEs and switch fabric; for example, if a router was processing traffic between web servers and the Internet, all of the processing would be performed by the forwarding plane.

Daemons

The Junos kernel has four major daemons; each of these daemons play a critical role within the MX and work together via Interprocess Communication (IPC) and routing sockets to communicate with the Junos kernel and other daemons. The following daemons take center stage and are required for the operation of Junos.

- Management daemon (mgd)
- Routing protocol daemon (rpd)
- Device control daemon (dcd)
- Chassis daemon (chassisd)

There are many more daemons for tasks such as NTP, VRRP, DHCP, and other technologies, but they play a smaller and more specific role in the software architecture.

Management Daemon

The Junos User Interface (UI) keeps everything in a centralized database. This allows Junos to handle data in interesting ways and open the door to advanced features such as configuration rollback, apply groups, and activating and deactivating entire portions of the configuration.

The UI has four major components: the configuration database, database schema, management daemon (mgd), and the command line interface (cli).

The management daemon (mgd) is the glue that holds the entire Junos User Interface (UI) together. At a high level, mgd provides a mechanism to process information for both network operators and daemons.

The interactive component of mgd is the Junos cli; this is a terminal-based application that allows the network operator an interface into Junos. The other side of mgd is the extensible markup language (XML) remote procedure call (RPC) interface; This provides an API through Junoscript and Netconf to allow for the development of automation applications.

The cli responsibilities are:

- Command-line editing
- Terminal emulation
- Terminal paging
- Displaying command and variable completions
- Monitoring log files and interfaces
- Executing child processes such as ping, traceroute, and ssh

mgd responsibilities include:

- Passing commands from the cli to the appropriate daemon
- Finding command and variable completions
- Parsing commands

It's interesting to note that the majority of the Junos operational commands use XML to pass data. To see an example of this, simply add the pipe command display xml to any command. Let's take a look at a simple command such as show isis adjacency.

```
{master}
dhanks@R1-RE0> show isis adjacency
Interface             System         L State        Hold (secs) SNPA
ae0.1                 R2-RE0         2  Up                   23
```

So far everything looks normal. Let's add the display xml to take a closer look.

```
{master}dhanks@R1-RE0> show isis adjacency | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/11.4R1/junos">
    <isis-adjacency-information xmlns="http://xml.juniper.net/junos/11.4R1/junos-
routing"
      junos:style="brief">
        <isis-adjacency>
            <interface-name>ae0.1</interface-name>
            <system-name>R2-RE0</system-name>
            <level>2</level>
            <adjacency-state>Up</adjacency-state>
            <holdtime>22</holdtime>
        </isis-adjacency>
    </isis-adjacency-information>
   <cli>
        <banner>{master}</banner>
    </cli>
</rpc-reply>
```

As you can see, the data is formatted in XML and received from mgd via RPC.

Routing Protocol Daemon

The routing protocol daemon (rpd) handles all of the routing protocols configured within Junos. At a high level, its responsibilities are receiving routing advertisements and updates, maintaining the routing table, and installing active routes into the forwarding table. In order to maintain process separation, each routing protocol configured on the system runs as a separate task within rpd. The other responsibility of rpd it to exchange information with the Junos kernel to receive interface modifications, send route information, and send interface changes.

Let's take a peek into rpd and see what's going on. The hidden command set task accounting toggles CPU accounting on and off. Use show task accounting to see the results.

```
{master}
dhanks@R1-RE0> set task accounting on
Task accounting enabled.
```

Now we're good to go. Junos is currently profiling daemons and tasks to get a better idea of what's using the routing engine CPU. Let's wait a few minutes for it to collect some data.

OK, let's check it out:

```
{master}
dhanks@R1-RE0> show task accounting
Task accounting is enabled.

Task                       Started    User Time  System Time  Longest Run
Scheduler                      265        0.003        0.000        0.000
Memory                           2        0.000        0.000        0.000
hakr                             1        0.000            0        0.000
ES-IS I/O./var/run/ppmd_c        6        0.000            0        0.000
IS-IS I/O./var/run/ppmd_c       46        0.000        0.000        0.000
```

```
PIM I/O./var/run/ppmd_con        9        0.000        0.000        0.000
IS-IS                           90        0.001        0.000        0.000
BFD I/O./var/run/bfdd_con        9        0.000            0        0.000
Mirror Task.128.0.0.6+598       33        0.000        0.000        0.000
KRT                             25        0.000        0.000        0.000
Redirect                         1        0.000        0.000        0.000
MGMT_Listen./var/run/rpd_        7        0.000        0.000        0.000
SNMP Subagent./var/run/sn       15        0.000        0.000        0.000
```

Not too much going on here, but you get the idea. Currently, running daemons and tasks within `rpd` are present and accounted for.

The `set task accounting` command is hidden for a reason. It's possible to put additional load on the Junos kernel while accounting is turned on. It isn't recommended to run this command on a production network unless instructed by JTAC. After your debugging is finished, don't forget to turn it back off with `set task accounting off`.

Don't forget to turn off accounting.

```
{master}
dhanks@R1-RE0> set task accounting off
Task accounting disabled.
```

Device Control Daemon

The device control daemon (`dcd`) is responsible for configuring interfaces based on the current configuration and available hardware. One feature of Junos is being able to configure nonexistent hardware, as the assumption is that the hardware can be added at a later date and "just work." An example is the expectation that you can configure `set interfaces ge-1/0/0.0 family inet address 192.168.1.1/24` and commit. Assuming there's no hardware in FPC1, this configuration will not do anything. As soon as hardware is installed into FPC1, the first port will be configured immediately with the address 192.168.1.1/24.

Chassis Daemon (and Friends)

The chassis daemon (`chassisd`) supports all chassis, alarm, and environmental processes. At a high level, this includes monitoring the health of hardware, managing a real-time database of hardware inventory, and coordinating with the alarm daemon (`alarmd`) and the craft daemon (`craftd`) to manage alarms and LEDs.

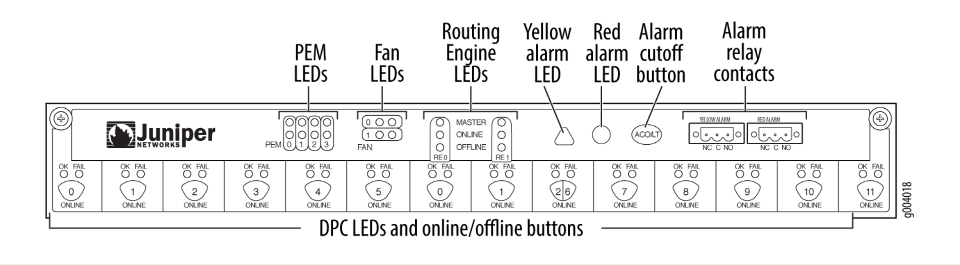


Figure 1-4. Juniper MX960 craft interface

It should all seem self-explanatory except for `craftd`; the craft interface that is the front panel of the device. Let's take a closer look at the MX960 craft interface.

The craft interfaces is a collection of buttons and LED lights to display the current status of the hardware and alarms. Information can also be obtained.

```
dhanks@R1-RE0> show chassis craft-interface

Front Panel System LEDs:
Routing Engine    0    1
--------------------------
OK                *    *
Fail              .    .
Master            *    .

Front Panel Alarm Indicators:
-----------------------------
Red LED      .
Yellow LED   .
Major relay  .
Minor relay  .

Front Panel FPC LEDs:
FPC    0   1   2
------------------
Red    .   .   .
Green  .   *   *

CB LEDs:
 CB    0   1
--------------
Amber  .   .
Green  *   *

PS LEDs:
  PS   0   1   2   3
--------------------
Red    .   .   .   .
Green  *   .   .   .

Fan Tray LEDs:
  FT   0
```

```
    ----------
    Red      .
    Green    *
```

One final responsibility of `chassisd` is monitoring the power and cooling environmentals. `chassisd` constantly monitors the voltages of all components within the chassis and will send alerts if the voltage crosses any thresholds. The same is true for the cooling. The chassis daemon constantly monitors the temperature on all of the different components and chips, as well as fan speeds. If anything is out of the ordinary, `chassisd` will create alerts. Under extreme temperature conditions, `chassisd` may also shut down components to avoid damage.

Routing Sockets

Routing sockets are a UNIX mechanism for controlling the routing table. The Junos kernel takes this same mechanism and extends it to include additional information to support additional attributes to create a carrier-class network operating system.

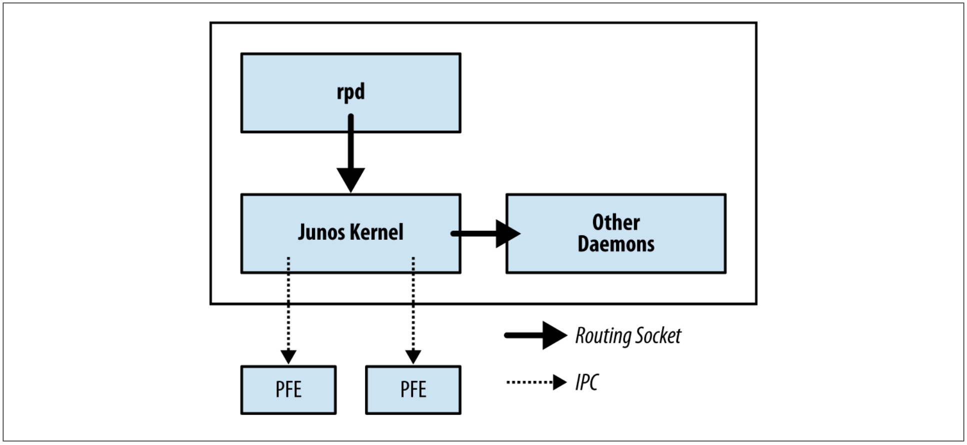


Figure 1-5. Routing socket architecture

At a high level, there are two actors when using routing sockets: state producer and state consumer. The `rpd` daemon is responsible for processing routing updates and thus is the state producer. Other daemons are considered state consumers because they process information received from the routing sockets.

Let's take a peek into the routing sockets and see what happens when we configure `ge-1/0/0.0` with an IP address of 192.168.1.1/24. Using the `rtsockmon` command from the shell will allow us to see the commands being pushed to the kernel from the Junos daemons.

```
    {master}
    dhanks@R1-RE0> start shell
    dhanks@R1-RE0% rtsockmon -st
```

```
        sender    flag   type       op
[16:37:52] dcd      P    iflogical  add     ge-1/0/0.0 flags=0x8000
[16:37:52] dcd      P    ifdev      change  ge-1/0/0 mtu=1514 dflags=0x3
[16:37:52] dcd      P    iffamily   add     inet mtu=1500 flags=0x8000000200000000
[16:37:52] dcd      P    nexthop    add     inet 192.168.1.255 nh=bcst
[16:37:52] dcd      P    nexthop    add     inet 192.168.1.0 nh=recv
[16:37:52] dcd      P    route      add     inet 192.168.1.255
[16:37:52] dcd      P    route      add     inet 192.168.1.0
[16:37:52] dcd      P    route      add     inet 192.168.1.1
[16:37:52] dcd      P    nexthop    add     inet 192.168.1.1 nh=locl
[16:37:52] dcd      P    ifaddr     add     inet local=192.168.1.1
[16:37:52] dcd      P    route      add     inet 192.168.1.1 tid=0
[16:37:52] dcd      P    nexthop    add     inet nh=rslv flags=0x0
[16:37:52] dcd      P    route      add     inet 192.168.1.0 tid=0
[16:37:52] dcd      P    nexthop    change  inet nh=rslv
[16:37:52] dcd      P    ifaddr     add     inet local=192.168.1.1 dest=192.168.1.0
[16:37:52] rpd      P    ifdest     change  ge-1/0/0.0, af 2, up, pfx 192.168.1.0/24
```

The author configured the interface ge-1/0/0 in a different terminal window and committed the change while the rtstockmon command was running.

The command rtsockmon is a Junos shell command that gives the user visibility into the messages being passed by the routing socket. The routing sockets are broken into four major components: sender, type, operation, and arguments. The sender field is used to identify which daemon is writing into the routing socket. The type identifies which attribute is being modified. The operation field is showing what is actually being performed. There are three basic operations: add, change, and delete. The last field is the arguments passed to the Junos kernel. These are sets of key and value pairs that are being changed.

In the previous example, you can see how dcd interacts with the routing socket to configure ge-1/0/0.0 and assign an IPv4 address.

- dcd creates a new logical interface (IFL).
- dcd changes the interface device (IFD) to set the proper MTU.
- dcd adds a new interface family (IFF) to support IPv4.
- dcd sets the nexthop, broadcast, and other attributes that are needed for the RIB and FIB.
- dcd adds the interface address (IFA) of 192.168.1.1.
- rpd finally adds a route for 192.168.1.1 and brings it up.

The rtsockmon command is only used to demonstrate the functionality of routing sockets and how daemons such as dcd and rpd use routing sockets to communicate routing changes to the Junos kernel.

Juniper MX Chassis

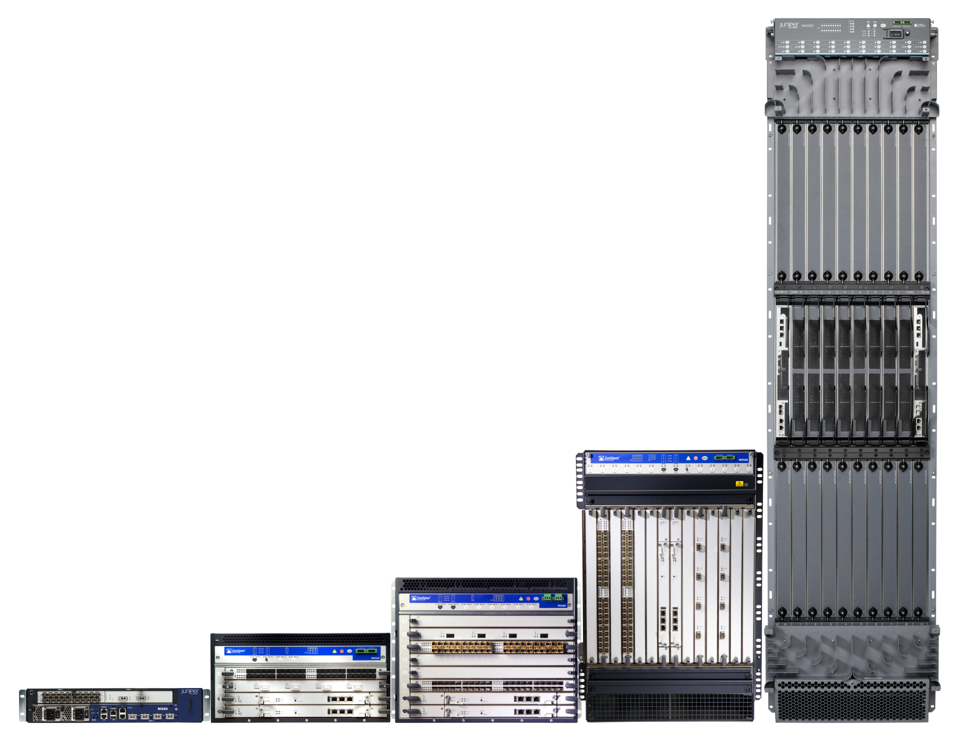


Figure 1-6. Juniper MX family

Ranging from 2U to 44U, the MX comes in many shapes and configurations. From left to right: MX80, MX240, MX480, MX960, and MX2020. The MX240 and higher models have chassis that house all components such as line cards, routing engines, and switching fabrics. The MX80 and below are considered midrange and only accept interface modules.

Table 1-2. Juniper MX Series capacity (Based on current hardware)

Model	DPC Capacity	MPC Capacity
MX80	N/A	80 Gbps
MX240	240 Gbps	1.280 Tbps
MX480	480 Gbps	3.84 Tbps
MX960	960 Gbps	7.68 Tbps
MX2020	N/A	12.8 Tbps

Please note that the DPC and MPC capacity is based on current hardware—4x10GE DPC and 32x10GE MPC—and is subject to change in the future as new hardware is released. This information only serves as an example.

The MX960 can accept up to 12 DPC line cards; using the 4x10GE DPC in the Table 1-2 calculations, you can come up with the following calculation:

40 Gbps * 2 (full-duplex) * 12 (maximum number of line cards on the MX960) = 960 Gbps.

Using the same calculation for the MPC capacity, the 32x10GE MPC line card can be used with the following calculation:

320 Gbps * 2 (full-duplex) * 12 (maximum number of line cards on the MX960) = 7.640 Tbps.

As the MX platform is upgraded with even faster switch fabrics and line cards, these calculations will change.

MX80

The MX journey begins with the MX80. It's a small, compact 2U router that comes in two models: the MX80 and MX80-48T. The MX80 supports two MICs, whereas the MX80-48T supports 48 10/100/1000BASE-T ports. Because of the small size of the MX80, all of the forwarding is handled by a single Trio chip and there's no need for a switch fabric. The added bonus is that in lieu of a switch fabric, each MX80 comes with four fixed 10GE ports.

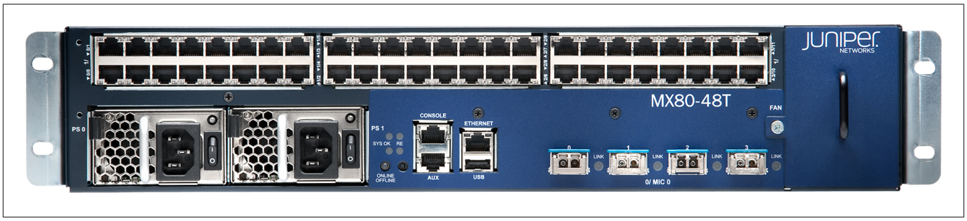


Figure 1-7. Juniper MX80-48T supports 48x1000BASE-T and 4x10GE ports

Each MX80 comes with field-replaceable, redundant power supplies and fan tray. The power supplies come in both AC and DC. Because the MX80 is so compact, it doesn't support slots for routing engines, Switch Control Boards (SCBs), or FPCs. The routing engine is built into the chassis and isn't replaceable. The MX80 only supports Modular Interface Cards (MICs).

The MX80 has a single routing engine and currently doesn't support features such as NSR, NSB, and ISSU.

But don't let the small size of the MX80 fool you. This is a true hardware-based router based on the Juniper Trio chipset. Here are some of the performance and scaling characteristics at a glance:

- 55 Mpps
- 1,000,000 IPv4 prefixes in the Forwarding Information Base (FIB)
- 4,000,000 IPv4 prefixes in the Routing Information Base (RIB)
- 16,000 logical interfaces (IFLs)
- 512,000 MAC addresses

MX80 Interface Numbering

The MX80 has two FPCs: FPC0 and FPC1. FPC0 will always be the four fixed 10GE ports located on the bottom right. The FPC0 ports are numbered from left to right starting with xe-0/0/0 and ending with xe-0/0/3.

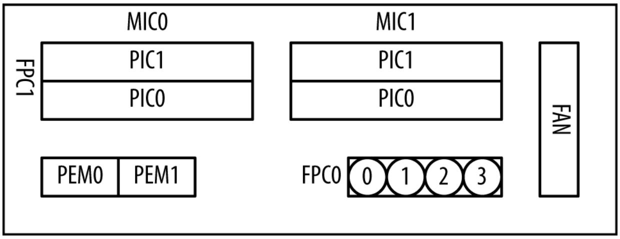


Figure 1-8. Juniper MX80 FPC and PIC locations

The dual power supplies are referred to as a Power Entry Module (PEM): PEM0 and PEM1.

FPC1 is where the MICs are installed. MIC0 is installed on the left side and MIC1 is installed on the right side. Each MIC has two Physical Interface Cards (PICs). Depending on the MIC, such as the 20x1GE or 2x10GE, the total number of ports will vary. Regardless of the number of ports, the port numbering is left to right and always begins with 0.

MX80-48T Interface Numbering

The MX80-48T interface numbering is very similar to the MX80. FPC0 remains the same and refers to the four fixed 10GE ports. The only difference is that FPC1 refers to the 48x1GE ports. FPC1 contains four PICs; the numbering begins at the bottom left, works its way up, and then shifts to the right starting at the bottom again. Each PIC contains 12x1GE ports numbered 0 through 11.

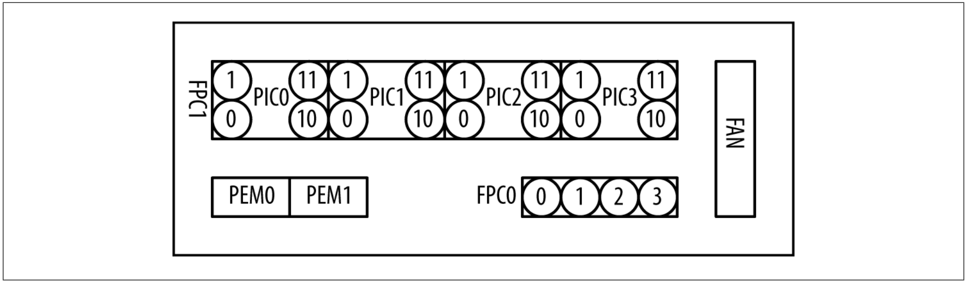


Figure 1-9. Juniper MX80-48T FPC and PIC location

Table 1-3. MX80-48T interface numbering

FPC	PIC	Interface Names
FPC0	PIC0	xe-0/0/0 through xe-0/0/3
FPC1	PIC0	ge-1/0/0 through ge-1/0/11
FPC1	PIC1	ge-1/1/0 through ge-1/1/11
FPC1	PIC2	ge-1/2/0 through ge-1/2/11
FPC1	PIC3	ge-1/3/0 through ge-1/3/11

With each PIC within FPC1 having 12x1GE ports and a total of four PICs, this brings the total to 48x1GE ports.

The MX80-48T has a fixed 48x1GE and 4x10GE ports and doesn't support MICs. These ports are tied directly to a single Trio chip as there is no switch fabric.

The MX80-48T doesn't have a Queuing Chip, thus doesn't support Hierarchical Class of Service (H-CoS). However, each port does support eight hardware queues and all other Junos Class of Service (CoS) features.

Midrange

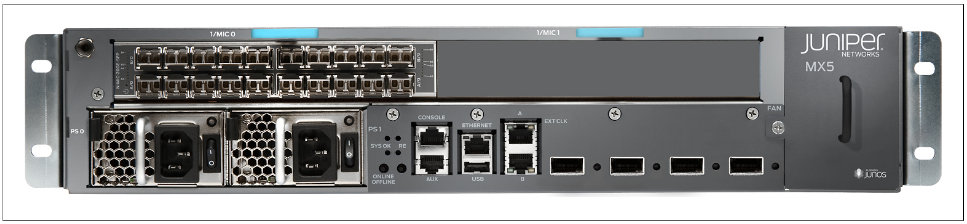


Figure 1-10. Juniper MX5.

If the MX80 is still too big of a router, there are a selection of licensing options to restrict the number of ports on the MX80. The benefit is that you get all of the performance and scaling of the MX80, but at a fraction of the cost. These licensing options are known as the MX Midrange: the MX5, MX10, MX40, and MX80.

Table 1-4. Midrange port restrictions

Model	MIC Slot 0	MIC Slot 1	Fixed 10GE Ports	Services MIC
MX5	Available	Restricted	Restricted	Available
MX10	Available	Available	Restricted	Available
MX40	Available	Available	Two ports available	Available
MX80	Available	Available	All four ports available	Available

Each router is software upgradable via a license. For example, the MX5 can be upgraded to the MX10 or directly to the MX40 or MX80.

When terminating a small number of circuits or Ethernet handoffs, the MX5 through the MX40 are the perfect choice. Although you're limited in the number of ports, all of the performance and scaling numbers are identical to the MX80. For example, given the current size of a full Internet routing table is about 420,000 IPv4 prefixes, the MX5 would be able to handle over nine full Internet routing tables.

Keep in mind that the MX5, MX10, and MX40 are really just an MX80. There is no difference in hardware, scaling, or performance. The only caveat is that the MX5, MX10, and MX40 use a different facia on the front of the router for branding.

There only restriction on the MX5, MX10, and MX40 are which ports are allowed to be configured. The software doesn't place any sort of bandwidth restrictions on the ports at all. There's a common misconception that the MX5 is a "5-gig router," but this isn't the case. For example, the MX5 comes with a 20x1GE MIC and is fully capable of running each port at line rate.

MX240

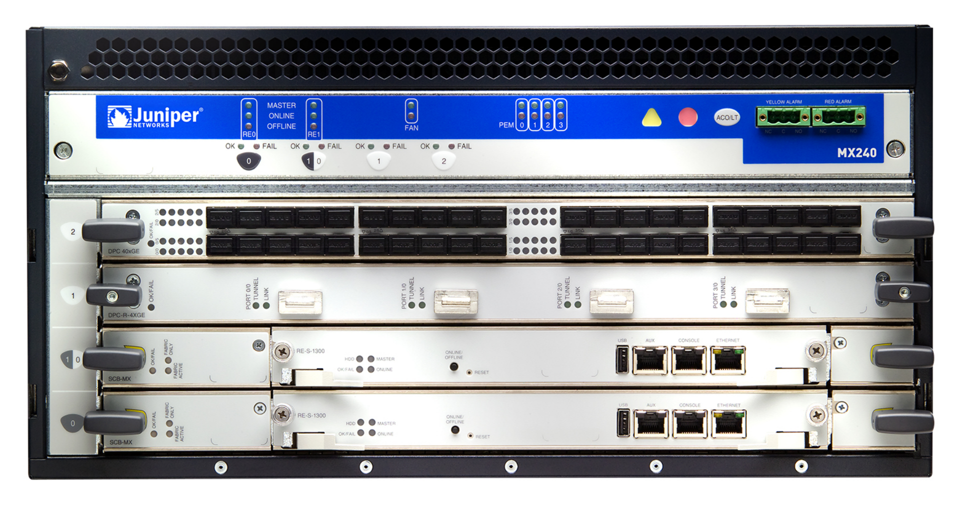


Figure 1-11. Juniper MX240

The MX240 is the first router in the lineup that has a chassis that supports modular routing engines, SCBs, and FPCs. The MX240 is 5U tall and supports four horizontal slots. There's support for one routing engine, or optional support for two routing engines. Depending on the number of routing engines, the MX240 supports either two or three FPCs.

The routing engine is installed into a SCB and will be described in more detail later in the chapter.

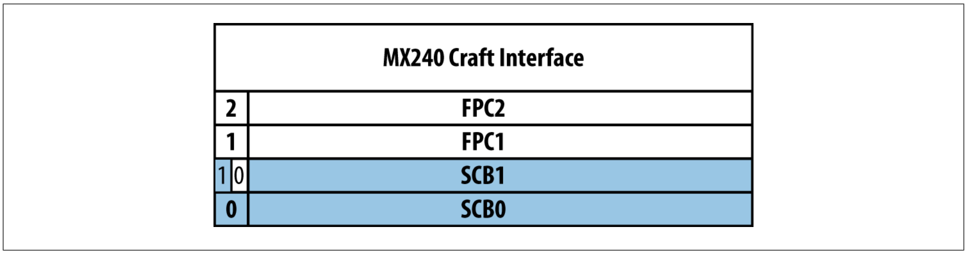
To support full redundancy, the MX240 requires two SCBs and routing engines. If a single SCB fails, there is enough switch fabric capacity on the other SCB to support the entire router at line rate. This is referred to as 1 + 1 SCB redundancy. In this configuration, only two FPCs are supported.

Alternatively, if redundancy isn't required, the MX240 can be configured to use a single SCB and routing engine. This configuration allows for three FPCs instead of two.

Interface Numbering

The MX240 is numbered from the bottom up starting with the SCB. The first SCB must be installed into the very bottom slot. The next slot up is a special slot that supports either a SCB or FPC, and thus begins the FPC numbering at 0. From there, you may install two additional FPCs as FPC1 and FPC2.

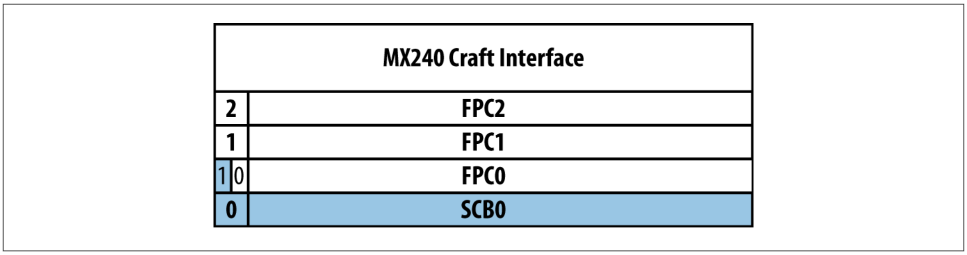
Full Redundancy. The SCBs must be installed into the very bottom slots to support 1 + 1 SCB redundancy. These slots are referred to as SCB0 and SCB1. When two SCBs are installed, the MX240 supports only two FPCs: FPC1 and FPC2.



	MX240 Craft Interface
2	FPC2
1	FPC1
1 0	SCB1
0	SCB0

Figure 1-12. Juniper MX240 interface numbering with SCB redundancy

No Redundancy. When a single SCB is used, it must be installed into the very bottom slot and obviously doesn't provide any redundancy; however, three FPCs are supported. In this configuration, the FPC numbering begins at FPC0 and ends at FPC2.



	MX240 Craft Interface
2	FPC2
1	FPC1
1 0	FPC0
0	SCB0

Figure 1-13. Juniper MX240 interface numbering without SCB redundancy

MX480

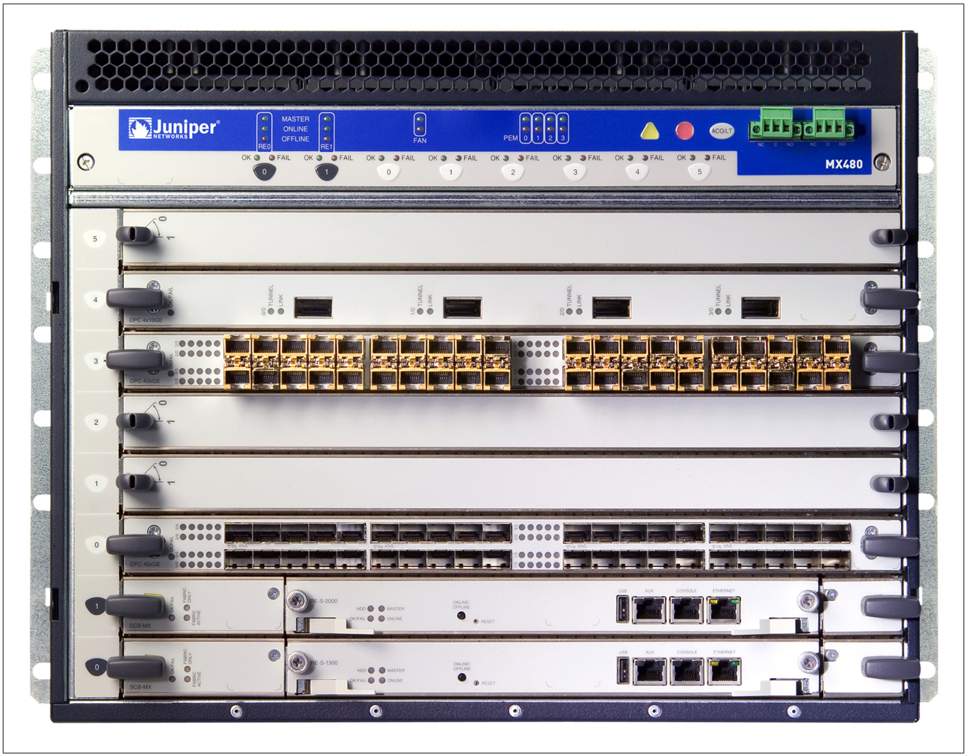


Figure 1-14. Juniper MX480

The MX480 is the big brother to the MX240. There are eight horizontal slots total. It supports two SCBs and routing engines as well as six FPCs in only 8U of space. The MX480 tends to be the most popular in enterprise because six slots is the “sweet spot.”

Like its little brother, the MX480 requires two SCBs and routing engines for full redundancy. If a single SCB were to fail, the other SCB would be able to support all six FPCs at line rate.

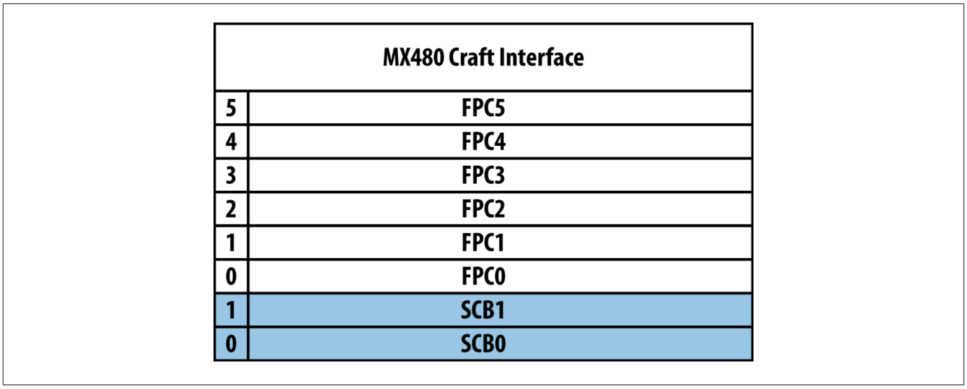
All components between the MX240 and MX480 are interchangeable. This makes the sparing strategy cost effective and provides FPC investment protection.

> There is custom keying on the SCB and FPC slots so that an SCB cannot be installed into a FPC slot and vice versa. In the case where the chassis supports either an SCB or FPC in the same slot, such as the MX240 or MX960, the keying will allow for both.

The MX480 is a bit different from the MX240 and MX960, as it has two dedicated SCB slots that aren't able to be shared with FPCs.

Interface Numbering

The MX480 is numbered from the bottom up. The SCBs are installed into the very bottom of the chassis into SCB0 and SCB1. From there, the FPCs may be installed and are numbered from the bottom up as well.



	MX480 Craft Interface
5	FPC5
4	FPC4
3	FPC3
2	FPC2
1	FPC1
0	FPC0
1	SCB1
0	SCB0

Figure 1-15. Juniper MX480 interface numbering with SCB redundancy

The MX480 slots are keyed specifically for two SCB and six FPC cards, while the MX240 and MX960 offer a single slot that's able to accept either SCB or FPC.

MX960

Some types of traffic require a big hammer. Enter the MX960. It's the sledgehammer of the MX Series. The MX960 is all about scale and performance. It stands at 16U and weighs in at 334lbs. The SCBs and FPCs are installed vertically into the chassis so that it can support 14 slots side to side.

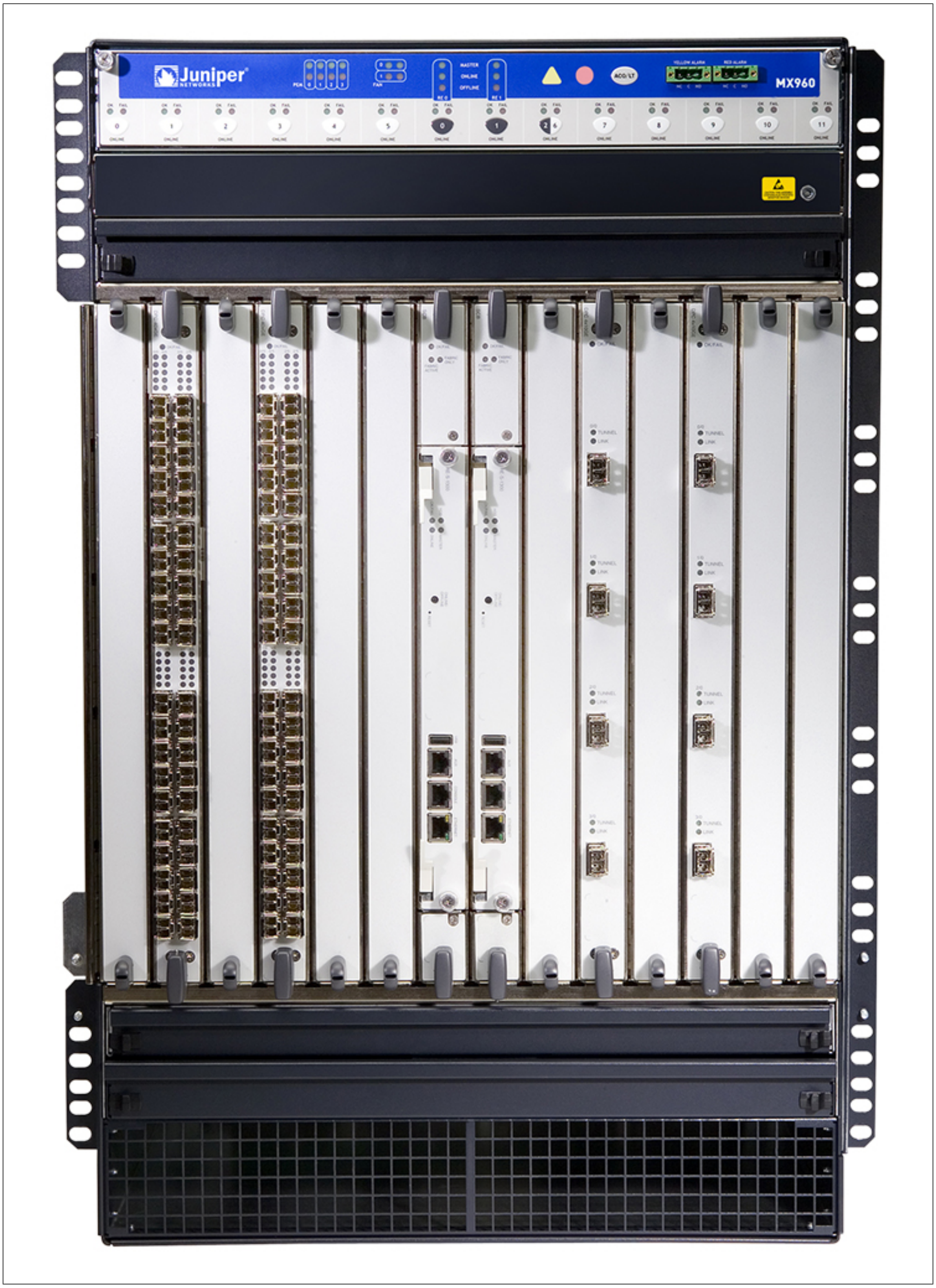


Figure 1-16. Juniper MX960

Because of the large scale, three SCBs are required for full redundancy. This is referred to as 2 + 1 SCB redundancy. If any SCB fails, the other two SCB are able to support all 11 FPCs at line rate.

If you like living life on the edge and don't need redundancy, the MX960 requires at least two SCBs to switch the available 12 FPCs.

The MX960 requires special power supplies that are not interchangeable with the MX240 or MX480.

Interface Numbering

The MX960 is numbered from the left to the right. The SCBs are installed in the middle, whereas the FPCs are installed on either side. Depending on whether or not you require SCB redundancy, the MX960 is able to support 11 or 12 FPCs.

Full Redundancy. The first six slots are reserved for FPCs and are numbered from left to right beginning at 0 and ending with 5. The next two slots are reserved and keyed for SCBs. The next slot is keyed for either a SCB or FPC. In the case of full redundancy, SCB2 needs to be installed into this slot. The next five slots are reserved for FPCs and begin numbering at 7 and end at 11.

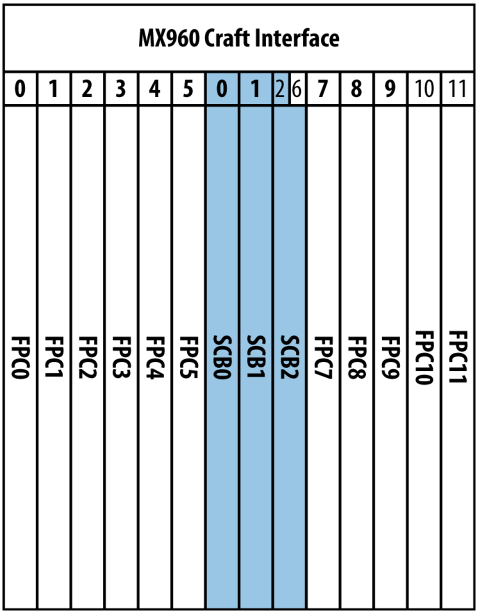


Figure 1-17. Juniper MX960 interface numbering with full 2 + 1 SCB redundancy

No Redundancy. Running with two SCBs gives you the benefit of being able to switch 12 FPCs at line rate. The only downside is that there's no SCB redundancy. Just like before, the first six slots are reserved for FPC0 through FPC5. The difference now is that SCB0 and SCB1 are to be installed into the next two slots. Instead of having SCB2, you install FPC6 into this slot. The remaining five slots are reserved for FPC7 through FPC11.

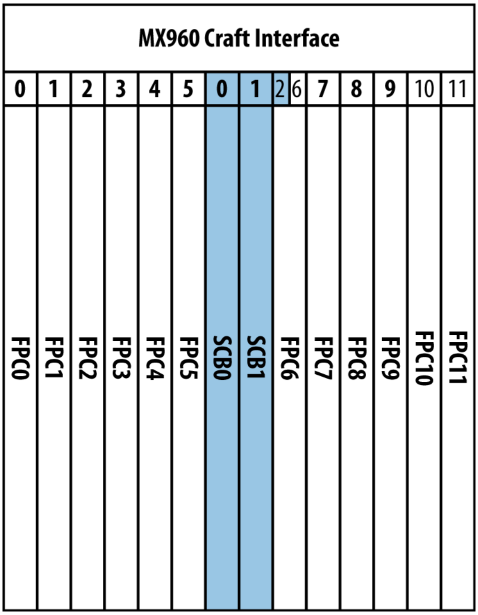


Figure 1-18. Juniper MX960 interface numbering without SCB redundancy

Trio

Juniper Networks prides itself on creating custom silicon and making history with silicon firsts. Trio is the latest milestone:

- 1998: First separation of control and data plane
- 1998: First implementation of IPv4, IPv6, and MPLS in silicon
- 2000: First line-rate 10 Gbps forwarding engine
- 2004: First multi-chassis router
- 2005: First line-rate 40 Gbps forwarding engine
- 2007: First 160 Gbps firewall
- 2009: Next generation silicon: Trio
- 2010: First 130 Gbps PFE; next generation Trio

Trio is a fundamental technology asset for Juniper that combines three major components: bandwidth scale, services scale, and subscriber scale. Trio was designed from the ground up to support high-density, line-rate 10G and 100G ports. Inline services such as IPFLOW, NAT, GRE, and BFD offer a higher level of quality of experience without requiring an additional services card. Trio offers massive subscriber scale in terms of logical interfaces, IPv4 and IPv6 routes, and hierarchical queuing.

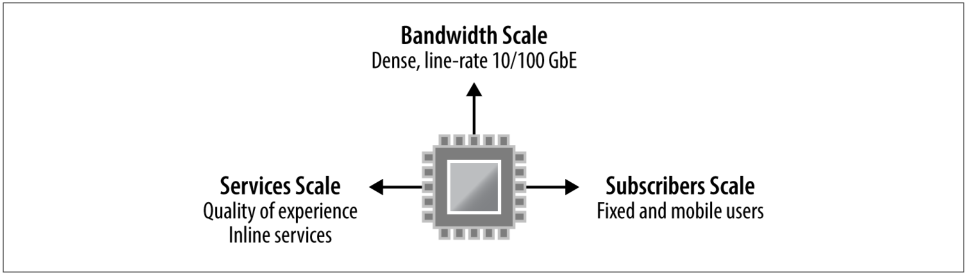


Figure 1-19. Juniper Trio scale: Services, bandwidth, and subscribers.

Trio is built upon a Network Instruction Set Processor (NISP). The key differentiator is that Trio has the performance of a traditional ASIC, but the flexibility of a field-programmable gate array (FPGA) by allowing the installation of new features via software. Here is just an example of the inline services available with the Trio chipset:

- Tunnel encapsulation and decapsulation
- IP Flow Information Export
- Network Address Translation
- Bidirectional Forwarding Detection
- Ethernet operations, administration, and management
- Instantaneous Link Aggregation Group convergence

Trio Architecture

The Trio chipset comprises of four major building blocks: Buffering, Lookup, Interfaces, and Dense Queuing.

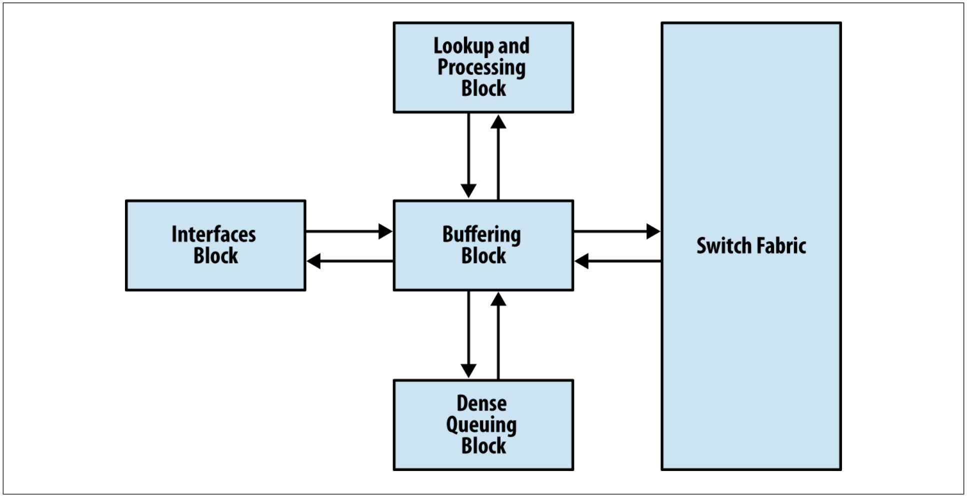


Figure 1-20. Trio functional blocks: Buffering, lookup, interfaces, and dense queuing

Each function is separated into its own block so that each function is highly optimized and cost efficient. Depending on the size and scale required, Trio is able to take these building blocks and create line cards that offer specialization such as hierarchical queuing or intelligent oversubscription.

Buffering Block

The Buffering Block ties together all of the other functional Trio blocks. It primarily manages packet data, fabric queuing, and revenue port queuing. The interesting thing to note about the Buffering Block is that it's possible to delegate responsibilities to other functional Trio blocks. As of the writing of this book, there are two primary use cases for delegating responsibility: process oversubscription and revenue port queuing.

In the scenario where the number of revenue ports on a single MIC is less than 24x1GE or 2x10GE, it's possible to move the handling of oversubscription to the Interfaces Block. This opens doors to creating oversubscribed line cards at an attractive price point that are able to handle oversubscription intelligently by allowing control plane and voice data to be processed during congestion.

The Buffering Block is able to process basic per port queuing. Each port has eight hardware queues, large delay buffers, and low latency queues (LLQs). If there's a requirement to have hierarchical class of service (H-QoS) and additional scale, this functionality can be delegated to the Dense Queuing Block.

Lookup Block

The Lookup Block has multi-core processors to support parallel tasks using multiple threads. This is the bread and butter of Trio. The Lookup Block supports all of the packet header processing such as:

- Route lookups
- MAC lookups
- Class of Service (QoS) Classification
- Firewall filters
- Policers
- Accounting
- Encapsulation
- Statistics

A key feature in the Lookup Block is that it supports Deep Packet Inspection (DPI) and is able to look over 256 bytes into the packet. This creates interesting features such as Distributed Denial of Service (DDoS) protection, which is covered in Chapter 4.

As packets are received by the Buffering Block, the packet headers are sent to the Lookup Block for additional processing. All processing is completed in one pass through the Lookup Block regardless of the complexity of the workflow. Once the Lookup Block has finished processing, it sends the modified packet headers back to the Buffering Block to send the packet to its final destination.

In order to process data at line rate, the Lookup Block has a large bucket of reduced-latency dynamic random access memory (RLDRAM) that is essential for packet processing.

Let's take a quick peek at the current memory utilization in the Lookup Block.

```
{master}
dhanks@R1-RE0> request pfe execute target fpc2 command "show jnh 0 pool usage"
SENT: Ukern command: show jnh 0 pool usage
GOT:
GOT: EDMEM overall usage:
GOT: [NH///|FW///|CNTR//////|HASH//////|ENCAPS////|--------------]
GOT: 0     2.0   4.0        9.0        16.8       20.9           32.0M
GOT:
GOT: Next Hop
GOT: [*************|-------] 2.0M (65% | 35%)
GOT:
GOT: Firewall
GOT: [|--------------------] 2.0M (1% | 99%)
GOT:
GOT: Counters
GOT: [|----------------------------------------] 5.0M (<1% | >99%)
GOT:
GOT: HASH
```

```
GOT: [*********************************************] 7.8M (100% | 0%)
GOT:
GOT: ENCAPS
GOT: [*****************************************] 4.1M (100% | 0%)
GOT:
LOCAL: End of file
```

The external data memory (EDMEM) is responsible for storing all of the firewall filters, counters, next-hops, encapsulations, and hash data. These values may look small, but don't be fooled. In our lab, we have an MPLS topology with over 2,000 L3VPNs including BGP route reflection. Within each VRF there is a firewall filter applied with two terms. As you can see, the firewall memory is barely being used. These memory allocations aren't static and are allocated as needed. There is a large pool of memory and each EDMEM attribute can grow as needed.

Interfaces Block

One of the optional components is the Interfaces Block. Its primary responsibility is to intelligently handle oversubscription. When using a MIC that supports less than 24x1GE or 2x10GE MACs, the Interfaces Block is used to manage the oversubscription.

> As new MICs are released, they may or may not have an Interfaces Block depending on power requirements and other factors. Remember that the Trio function blocks are like building blocks and some blocks aren't required to operate.

Each packet is inspected at line rate, and attributes such as Ethernet Type Codes, Protocol, and other Layer 4 information are used to evaluate which buffers to enqueue the packet towards the Buffering Block. Preclassification allows the ability to drop excess packets as close to the source as possible, while allowing critical control plane packets through to the Buffering Block.

There are four queues between the Interfaces and Buffering Block: real-time, control traffic, best effort, and packet drop. Currently, these queues and preclassifications are not user configurable; however, it's possible to take a peek at them.

Let's take a look at a router with a 20x1GE MIC that has an Interfaces Block.

```
dhanks@MX960> show chassis hardware
Hardware inventory:
Item             Version  Part number  Serial number     Description
Chassis                                JN10852F2AFA      MX960
Midplane         REV 02   710-013698   TR0019            MX960 Backplane
FPM Board        REV 02   710-014974   JY4626            Front Panel Display
Routing Engine 0 REV 05   740-031116   9009066101        RE-S-1800x4
Routing Engine 1 REV 05   740-031116   9009066210        RE-S-1800x4
CB 0             REV 10   750-031391   ZB9999            Enhanced MX SCB
CB 1             REV 10   750-031391   ZC0007            Enhanced MX SCB
CB 2             REV 10   750-031391   ZC0001            Enhanced MX SCB
```

```
FPC 1            REV 28   750-031090   YL1836            MPC Type 2 3D EQ
  CPU            REV 06   711-030884   YL1418            MPC PMB 2G
  MIC 0          REV 05   750-028392   JG8529            3D 20x 1GE(LAN) SFP
  MIC 1          REV 05   750-028392   JG8524            3D 20x 1GE(LAN) SFP
```

We can see that FPC1 supports two 20x1GE MICs. Let's take a peek at the preclassification on FPC1.

```
dhanks@MX960> request pfe execute target fpc1 command "show precl-eng summary"
SENT: Ukern command: show precl-eng summary
GOT:
GOT:  ID  precl_eng name       FPC PIC   (ptr)
GOT: --- -------------------- ---- ---  --------
GOT:   1 IX_engine.1.0.20       1    0  442484d8
GOT:   2 IX_engine.1.1.22       1    1  44248378
LOCAL: End of file
```

It's interesting to note that there are two preclassification engines. This makes sense as there is an Interfaces Block per MIC. Now let's take a closer look at the preclassification engine and statistics on the first MIC.

```
dhanks@MX960> request pfe execute target fpc1 command "show precl-eng 1 statistics"
SENT: Ukern command: show precl-eng 1 statistics
GOT:
GOT:          stream    Traffic
GOT:  port      ID       Class         TX pkts         RX pkts        Dropped pkts
GOT: ------  -------  ----------       ---------       ------------   ------------
GOT:   00       1025      RT          000000000000    000000000000   000000000000
GOT:   00       1026      CTRL        000000000000    000000000000   000000000000
GOT:   00       1027      BE          000000000000    000000000000   000000000000
```

Each physical port is broken out and grouped by traffic class. The number of packets dropped is maintained in a counter on the last column. This is always a good place to look if the router is oversubscribed and dropping packets.

Let's take a peek at a router with a 4x10GE MIC that doesn't have an Interfaces Block.

```
{master}
dhanks@R1-RE0> show chassis hardware
Hardware inventory:
Item             Version  Part number  Serial number     Description
Chassis                                JN111992BAFC      MX240
Midplane         REV 07   760-021404   TR5026            MX240 Backplane
FPM Board        REV 03   760-021392   KE2411            Front Panel Display
Routing Engine 0 REV 07   740-013063   1000745244        RE-S-2000
Routing Engine 1 REV 06   740-013063   1000687971        RE-S-2000
CB 0             REV 03   710-021523   KH6172            MX SCB
CB 1             REV 10   710-021523   ABBM2781          MX SCB
FPC 2            REV 25   750-031090   YC5524            MPC Type 2 3D EQ
  CPU            REV 06   711-030884   YC5325            MPC PMB 2G
  MIC 0          REV 24   750-028387   YH1230            3D 4x 10GE  XFP
  MIC 1          REV 24   750-028387   YG3527            3D 4x 10GE  XFP
```

Here we can see that FPC2 has two 4x10GE MICs. Let's take a closer look and the preclassification engines.

```
{master}
dhanks@R1-RE0> request pfe execute target fpc2 command "show precl-eng summary"
SENT: Ukern command: show precl-eng summary
GOT:
GOT:  ID  precl_eng name       FPC PIC   (ptr)
GOT: --- -------------------- ---- ---  --------
GOT:   1 MQ_engine.2.0.16       2   0  435e2318
GOT:   2 MQ_engine.2.1.17       2   1  435e21b8
LOCAL: End of file
```

The big difference here is the preclassification engine name. Previously, it was listed as “IX_engine” with MICs that support an Interfaces Block. MICs such as the 4x10GE do not have an Interfaces Block, so the preclassification is performed on the Buffering Block, or, as listed here, the “MQ_engine.”

> The author has used hidden commands to illustrate the roles and responsibilities of the Interfaces Block. Caution should be used when using these commands as they aren’t supported by Juniper.

The Buffering Block’s WAN interface can operate either in MAC mode or in the Universal Packet over HSL2 (UPOH) mode. This creates a difference in operation between the MPC1 and MPC2 line cards. The MPC1 only has a single Trio chipset, thus only MICs that can operate in MAC mode are compatible with this line card. On the other hand, the MPC2 has two Trio chipsets. Each MIC on the MPC2 is able to operate in either mode, thus compatible with more MICs. This will be explained in more detail later in the chapter.

Dense Queuing Block

Depending on the line card, Trio offers an optional Dense Queuing Block that offers rich Hierarchical QoS that supports up to 512,000 queues with the current generation of hardware. This allows for the creation of schedulers that define drop characteristics, transmission rate, and buffering that can be controlled separately and applied at multiple levels of hierarchy.

The Dense Queuing Block is an optional functional Trio block. The Buffering Block already supports basic per port queuing. The Dense Queuing Block is only used in line cards that require H-QoS or additional scale beyond the Buffering Block.

Line Cards and Modules

To provide the high-density and high-speed Ethernet services, a new type of Flexible Port Concentrator (FPC) had to be created called the Dense Port Concentrator (DPC). This first-generation line card allowed up to 80 Gbps ports per slot.

The DPC line cards utilize a previous ASIC from the M series called the I-CHIP. This allowed Juniper to rapidly build the first MX line cards and software.

The Modular Port Concentrator (MPC) is the second-generation line card created to further increase the density to 160 Gbps ports per slot. This generation of hardware is created using the Trio chipset. The MPC supports MICs that allow you to mix and match different modules on the same MPC.

Table 1-5. Juniper MX line card and module types

FPC Type/Module Type	Description
Dense Port Concentrator (DPC)	First-generation high-density and high-speed Ethernet line cards
Modular Port Concentrator (MPC)	Second-generation high-density and high-speed Ethernet line cards supporting modules
Module Interface Card (MIC)	Second-generation Ethernet and optical modules that are inserted into MPCs

It's a common misconception that the "modular" part of MPC derives its name only from its ability to accept different kinds of MICs. This is only half of the story. The MPC also derives its name from being able to be flexible when it comes to the Trio chipset. For example, the MPC-3D-16x10GE-SFPP line card is a fixed port configuration, but only uses the Buffering Block and Lookup Block in the PFE complex. As new line cards are introduced in the future, the number of fundamental Trio building blocks will vary per card as well, thus living up to the "modular" name.

Dense Port Concentrator

The DPC line cards come in six different models to support varying different port configurations. There's a mixture of 1G, 10G, copper, and optical. There are three DPC types: routing and switching (DPCE-R), switching (DPCE-X), and enhanced queuing (DPCE-Q).

The DPCE-R can operate at either Layer 3 or as a pure Layer 2 switch. It's generally the most cost-effective when using a sparing strategy for support. The DPCE-R is the most popular choice as it supports very large route tables and can be used in a pure switching configuration as well.

The DPCE-X has the same features and services as the DPCE-R; the main difference is that the route table is limited to 32,000 prefixes and cannot use L3VPNs on this DPC. These line cards make sense when being used in a very small environment or in a pure Layer 2 switching scenario.

The DPCE-Q supports all of the same features and services as the DPCE-R and adds additional scaling around H-QoS and number of queues.

Table 1-6. DPC line card types

Model	DPCE-R	DPCE-X	DPCE-Q
40x1GE SFP	Yes	Yes	Yes

Model	DPCE-R	DPCE-X	DPCE-Q
40x1GE TX	Yes	Yes	No
20x1GE SFP	No	No	Yes
4x10GE XFP	Yes	Yes	Yes
2x10GE XFP	Yes	No	No
20x1GE and 2x10GE	Yes	Yes	Yes

The DPC line cards are still supported, but there is no active development of new features being brought to these line cards. For new deployments, it's recommended to use the newer, second-generation MPC line cards. The MPC line cards use the Trio chipset and are where Juniper is focusing all new features and services.

Modular Port Concentrator

The MPC line cards are the second generation of line cards for the MX. There are two significant changes when moving from the DPC to MPC: chipset and modularity. All MPCs are now using the Trio chipset to support more scale, bandwidth, and services. The other big change is that now the line cards are modular using MICs.

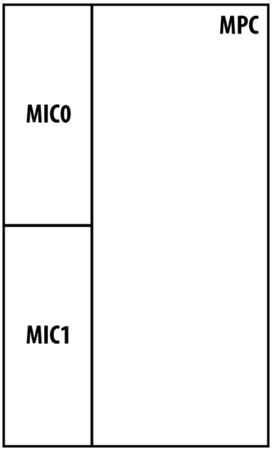


Figure 1-21. High-level architecture of MPCs and MICs

The MPC can be thought of as a type of intelligent shell or carrier for MICs. This change in architecture allows the separation of physical ports, oversubscription, features, and services. All of the oversubscription, features, and services are managed within the MPC. Physical port configurations are isolated to the MIC. This allows the same MIC to be used in many different types of MPCs depending on the number of features and scale required.

As of Junos 11.4, there are three different categories of MPCs. Each model has a different number of Trio chipsets providing different options of scaling and bandwidth.

Table 1-7. Modular port concentrator models

Model	# of Trio chipsets	Trio Bandwidth	Interface Support
MPC1	1	40 Gbps	1GE and 10GE
MPC2	2	80 Gbps	1GE and 10GE
MPC3E	1	130 Gbps	1GE, 10GE, 40GE, and 100GE

The MPC3 is the first of many more MPCs that will use an enhanced Trio chipset that is designed to support 40G and 100G interfaces. The MPC3 was designed to be similar to the MPC1 architecture whereby a single Trio chipset handles both MICs and is intentionally oversubscribed to offer an attractive price point.

> It's important to note that the MPC bandwidth listed previously represents current-generation hardware that's available as of the writing of this book and is subject to change with new software and hardware releases.

Similar to the first-generation DPC line cards, the MPC line cards also support the ability to operate in Layer 2, Layer 3, or enhanced queuing modes. This allows you choose only the features and services required.

Table 1-8. MPC feature matrix

Model	Full Layer 2	Full Layer 3	Enhanced Queuing
MX-3D	Yes	No	No
MX-3D-Q	Yes	No	Yes
MX-3D-R-B	Yes	Yes	No
MX-3D-Q-R-B	Yes	Yes	Yes

Most Enterprise customers tend to choose the MX-3D-R-B model as it supports both Layer 2 and Layer 3. Typically, there's no need for enhanced queuing or scale when building a data center. Most Service Providers prefer to use the MX-3D-Q-R-B as it provides both Layer 2 and Layer 3 services in addition to enhanced queuing. A typical use case for a Service Provider is having to manage large routing tables, many customers, and provide H-QoS to enforce customer service level agreements (SLAs).

- The MX-3D-R-B is the most popular choice, as it offers full Layer 3 and Layer 2 switching support.

- The MX-3D has all of the same features and services as the MX-3D-R-B but has limited Layer 3 scaling. When using BGP or an IGP, the routing table is limited to 32,000 routes. The other restriction is that MPLS L3VPNs cannot be used on these line cards.
- The MX-3D-Q has all of the same features, services, and reduced Layer 3 capacity as the MX-3D, but offers enhanced queuing. This adds the ability to configure H-QoS and increase the scale of queues.
- The MX-3D-Q-R-B combines all of these features together to offer full Layer 2, Layer 3, and enhanced queuing together in one line card.

MPC1

Let's revisit the MPC models in more detail. The MPC starts off with the MPC1, which has a single Trio chipset. The use case for this MPC is to offer an intelligently oversubscribed line card for an attractive price. All of the MICs that are compatible with the MPC1 have the Interfaces Block built into the MIC to handle oversubscription.

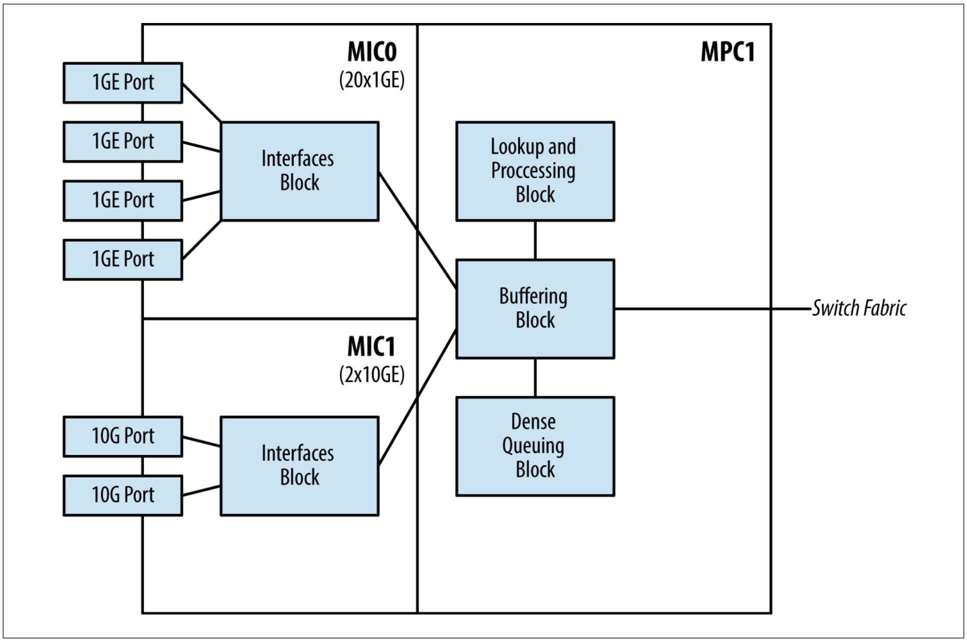


Figure 1-22. MPC1 architecture

With the MPC1, the single Trio chipset handles both MICs. Each MIC is required to share the bandwidth that's provided by the single Trio chipset, thus the Interfaces Block is delegated to each MIC to intelligently handle oversubscription. Because each Trio chipset can only operate in MAC mode or UPOH mode, the MPC1 must operate in

MAC mode to be able to support the 20x1GE and 2x10GE MICs. Unfortunately, the 4x10GE MIC only operates in UPOH mode and isn't compatible with the MPC1.

MPC2

The MPC2 is very similar in architecture to the MPC1, but adds an additional Trio chipset for a total count of two.

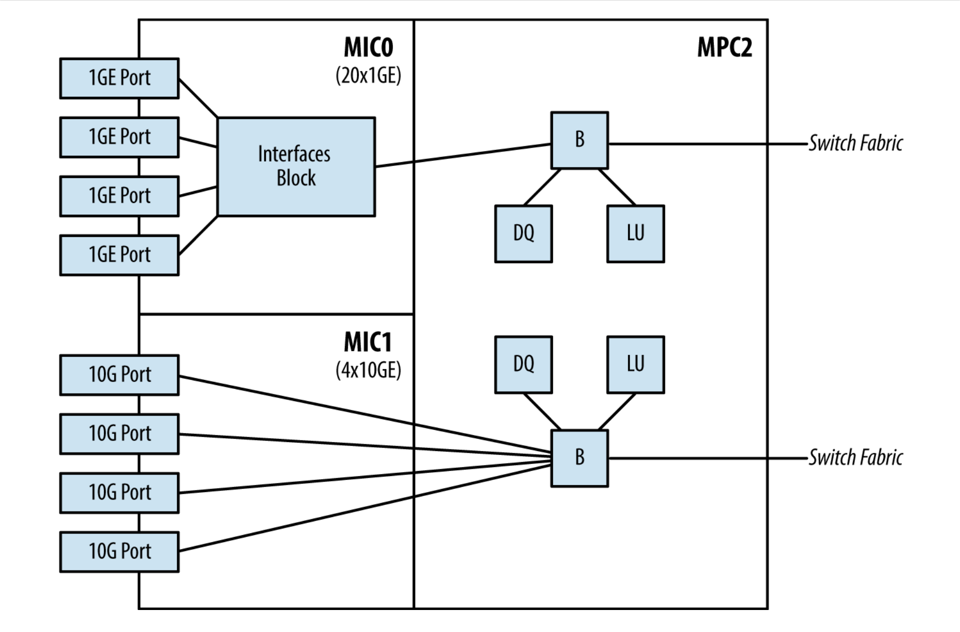


Figure 1-23. MPC2 architecture. B = Buffering Block, LU = LookUp Block, and DQ = Dense Queuing Block

The MPC2 offers a dedicated Trio chipset per MIC, effectively doubling the bandwidth and scaling from the previous MPC1. In the MPC2 architecture example, it's possible to combine MICs such as the 2x10GE and 4x10GE. Figure 1-23 illustrates the MPC2 being able to operate in both MAC mode and UPOH mode. Please note that Figure 1-23 uses several abbreviations:

- B = Buffering Block
- LU = Lookup Block
- DQ = Dense Queuing Block

The 2x10GE MIC is designed to operate in both the MPC1 and MPC2 and thus has an Interfaces Block to handle oversubscription. In the case of the 4x10GE MIC, it's designed to only operate in the MPC2 and thus doesn't require an Interfaces Block as it ties directly into a dedicated Buffering Block.

MPC-3D-16X10GE-SFPP

The MPC-3D-16X10GE-SFPP is a full-width line card that doesn't support any MICs. However, it does support 16 fixed 10G ports. This MPC is actually one of the most popular MPCs because of the high 10G port density and offers the lowest price per 10G port.

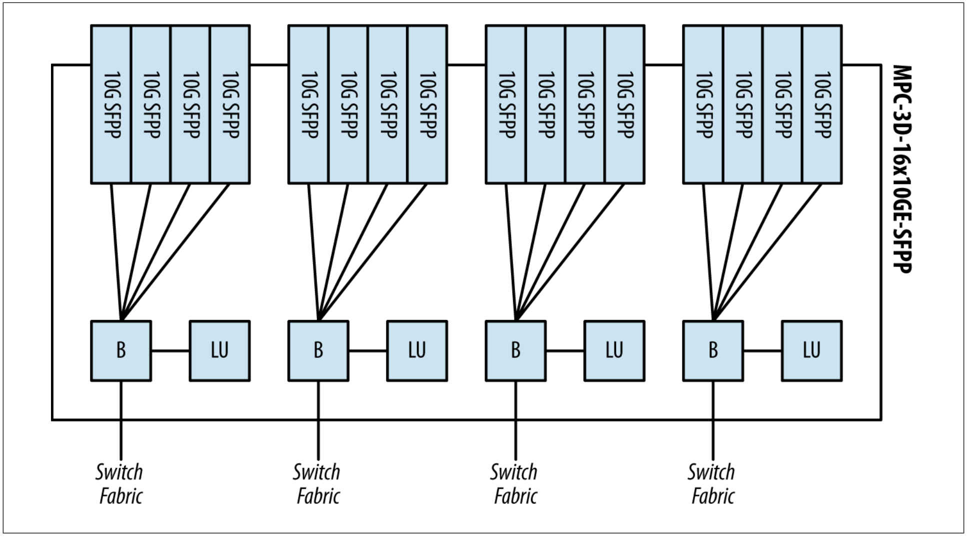


Figure 1-24. High-level architecture of MPC-3D-16x10GE-SFPP

The MPC-3D-16X10GE-SFPP has four Trio chipsets equally divided between the 16 ports. This allows each group of 4x10G interfaces to have a dedicated Trio chipset. This enables the MPC-3D-16X10GE-SFPP to operate at line rate for all ports.

If you're ever curious how many PFEs are on a FPC, you can use the `show chassis fabric map` command.

First, let's find out which FPC the MPC-3D-16X10GE-SFPP is installed into.

```
dhanks@MX960> show chassis hardware | match 16x
FPC 3            REV 23   750-028467   YJ2172           MPC 3D 16x 10GE
```

We found what we were looking for. The MPC-3D-16X10GE-SFPP is installed into FPC3. Now let's take a peek at the fabric map and see which links are `Up`, thus detecting the presence of PFEs within FPC3.

```
dhanks@MX960> show chassis fabric map | match DPC3
DPC3PFE0->CB0F0_04_0    Up         CB0F0_04_0->DPC3PFE0    Up
DPC3PFE1->CB0F0_04_1    Up         CB0F0_04_1->DPC3PFE1    Up
DPC3PFE2->CB0F0_04_2    Up         CB0F0_04_2->DPC3PFE2    Up
DPC3PFE3->CB0F0_04_3    Up         CB0F0_04_3->DPC3PFE3    Up
DPC3PFE0->CB0F1_04_0    Up         CB0F1_04_0->DPC3PFE0    Up
DPC3PFE1->CB0F1_04_1    Up         CB0F1_04_1->DPC3PFE1    Up
DPC3PFE2->CB0F1_04_2    Up         CB0F1_04_2->DPC3PFE2    Up
```

```
DPC3PFE3->CB0F1_04_3    Up         CB0F1_04_3->DPC3PFE3    Up
DPC3PFE0->CB1F0_04_0    Up         CB1F0_04_0->DPC3PFE0    Up
DPC3PFE1->CB1F0_04_1    Up         CB1F0_04_1->DPC3PFE1    Up
DPC3PFE2->CB1F0_04_2    Up         CB1F0_04_2->DPC3PFE2    Up
DPC3PFE3->CB1F0_04_3    Up         CB1F0_04_3->DPC3PFE3    Up
DPC3PFE0->CB1F1_04_0    Up         CB1F1_04_0->DPC3PFE0    Up
DPC3PFE1->CB1F1_04_1    Up         CB1F1_04_1->DPC3PFE1    Up
DPC3PFE2->CB1F1_04_2    Up         CB1F1_04_2->DPC3PFE2    Up
DPC3PFE3->CB1F1_04_3    Up         CB1F1_04_3->DPC3PFE3    Up
```

That wasn't too hard. The only tricky part is that the output of the `show chassis fabric command` still lists the MPC as DPC in the output. No worries, we can perform a match for DPC3. As we can see, the MPC-3D-16X10GE-SFPP has a total of four PFEs, thus four Trio chipsets. Note that `DPC3PFE0` through `DPC3PFE3` are present and listed as `Up`. This indicates that the line card in FPC3 has four PFEs.

The MPC-3D-16X10GE-SFPP doesn't support H-QoS because there's no Dense Queuing Block. This leaves only two functional Trio blocks per PFE on the MPC-3D-16X10GE-SFPP: the Buffering Block and Lookup Block.

Let's verify this by taking a peek at the preclassification engine:

```
dhanks@MX960> request pfe execute target fpc3 command "show precl-eng summary"
SENT: Ukern command: show prec sum
GOT:
GOT:  ID  precl_eng name       FPC PIC   (ptr)
GOT: --- -------------------- ---- ---  --------
GOT:   1 MQ_engine.3.0.16       3   0  4837d5b8
GOT:   2 MQ_engine.3.1.17       3   1  4837d458
GOT:   3 MQ_engine.3.2.18       3   2  4837d2f8
GOT:   4 MQ_engine.3.3.19       3   3  4837d198
LOCAL: End of file
```

As expected, the Buffering Block is handling the preclassification. It's interesting to note that this is another good way to see how many Trio chipsets are inside of a FPC. The preclassifications engines are listed ID 1 through 4 and match our previous calculation using the `show chassis fabric map` command.

MPC3E

The MPC3E is the first modular line card for the MX Series to accept 100G and 40G MICs. It's been designed from the ground up to support interfaces beyond 10GE, but also remains compatible with some legacy MICs.

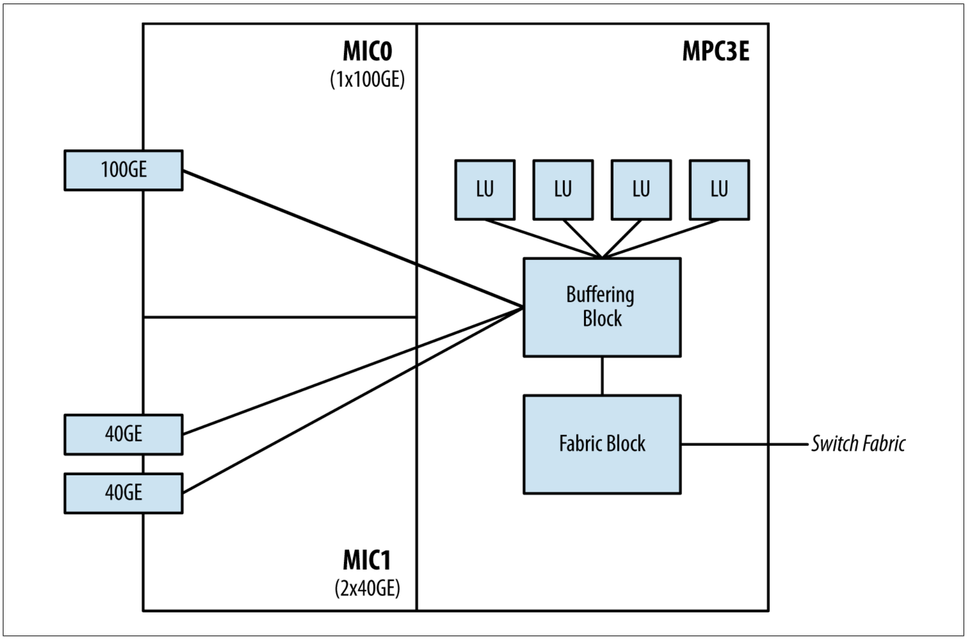


Figure 1-25. MPC3E architecture

There are several new and improved features on the MPC3E. The most notable is that the Buffering Block has been increased to support 130 Gbps and number of Lookup Blocks has increased to four in order to support 100GE interfaces. The other major change is that the fabric switching functionality has been moved out of the Buffering Block and into a new Fabric Functional Block.

The MPC3E can provide line-rate performance for a single 100GE interface; otherwise it's known that this line card is oversubscribed 1.5:1. For example, the MPC3E can support 2x100GE interfaces, but the Buffering Block can only handle 130Gbps. This can be written as 200:130, or roughly 1.5:1 oversubscription.

Enhanced Queuing isn't supported on the MPC3E due to the lack of a Dense Queuing Block. However, this doesn't mean that the MPC3E isn't capable of class of service. The Buffering Block, just like the MPC-3D-16x10GE-SFPP, is capable of basic port-level class of service.

Multiple Lookup Block Architecture. All MPC line cards previous to the MPC3E had a single Lookup Block per Trio chipset; thus, no Lookup Block synchronization was required. The MPC3E is the first MPC to introduce multiple Lookup Blocks. This creates an interesting challenge in synchronizing the Lookup Block operations.

In general, the Buffering Block will spray packets across all Lookup Blocks in a round-robin fashion. This means that a particular traffic flow will be processed by multiple Lookup Blocks.

Source MAC Learning. At a high level, the MPC3E learns the source MAC address from the WAN ports. One of the four Lookup Blocks is designated as the master and the three remaining Lookup blocks are designated as the slaves.

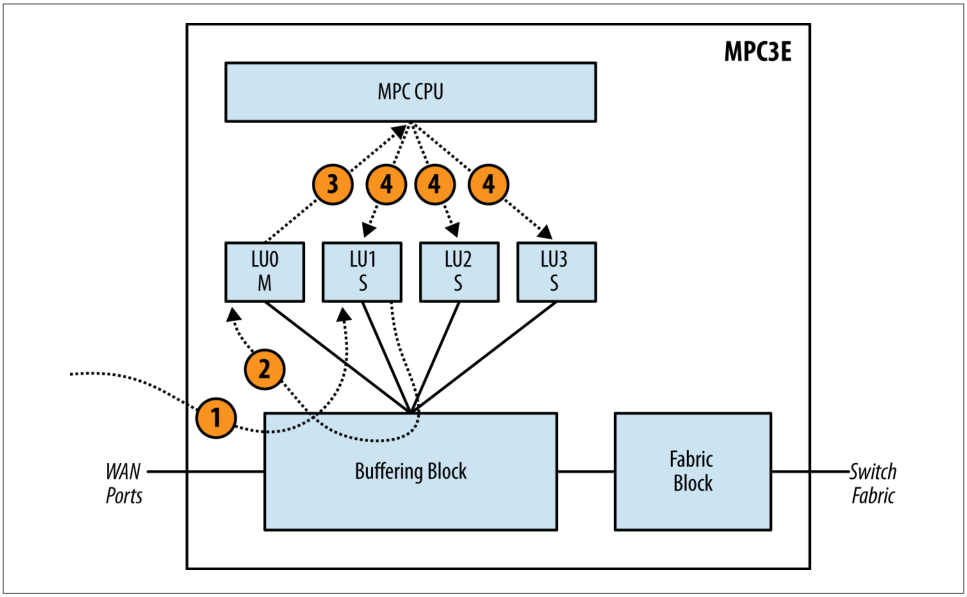


Figure 1-26. MPC3E source MAC learning

The Master Lookup Block is responsible for updating the other Slave Lookup Blocks. Figure 1-26 illustrates the steps taken to synchronize all of the Lookup Blocks.

1. The packet enters the Buffering Block and happens to be sprayed to LU1, which is designated as a Slave Lookup Block.
2. LU1 updates its own table with the source MAC address. It then notifies the Master Lookup Block LU0. The update happens via the Buffering Block to reach LU0.
3. The Master Lookup Block LU0 receives the source MAC address update and updates its local table accordingly. LU0 sends the source MAC address update to the MPC CPU.
4. The MPC CPU receives the source MAC address update and in turn updates all Lookup Blocks in parallel.

Destination MAC Learning. The MPC3E learns destination MAC addresses based off the packet received from other PFEs over the switch fabric. Unlike the source MAC learning, there's no concept of a master or slave Lookup Block.

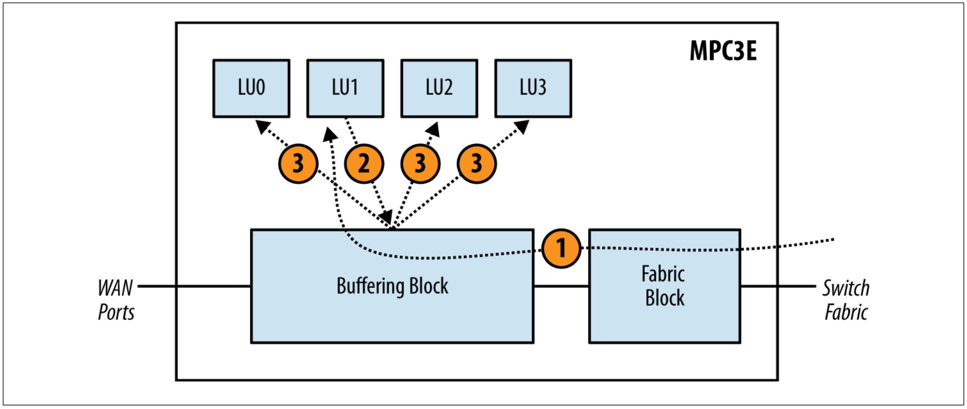


Figure 1-27. MPC3E destination MAC learning

The Lookup Block that receives the packet from the switch fabric is responsible for updating the other Lookup Blocks. Figure 1-27 illustrates how destination MAC addresses are synchronized:

1. The packet enters the Fabric Block and Buffering Block. The packet happens to be sprayed to LU1. LU1 updates its local table.
2. LU1 then sends updates to all other Lookup Blocks via the Buffering Block.
3. The Buffering Block takes the update from LU1 and then updates the other Lookup Blocks in parallel. As each Lookup Block receives the update, the local table is updated accordingly.

Policing. Recall that the Buffering Block on the MPC3E sprays packets across Lookup Blocks evenly, even for the same traffic flow. Statistically, each Lookup Block receives about 25% of all traffic. When defining and configuring a policer, the MPC3E must take the bandwidth and evenly distribute it among the Lookup Blocks. Thus each Lookup Block is programmed to police 25% of the configured policer rate. Let's take a closer look:

```
firewall {
    policer 100M {
        if-exceeding {
            bandwidth-limit 100m;
            burst-size-limit 6250000;
        }
        then discard;
    }
}
```

The example policer `100M` is configured to enforce a `bandwidth-limit` of `100m`. In the case of the MPC3E, each Lookup Block will be configured to police `25m`. Because packets are statistically distributed round-robin to all four Lookup blocks evenly, the aggregate will equal the original policer `bandwidth-limit` of `100m`. `25m` * 4 (Lookup Blocks) = `100m`.

Packet Walkthrough

Now that you have an understanding of the different Trio functional blocks and the layout of each line card, let's take a look at how a packet is processed through each of the major line cards. Because there are so many different variations of functional blocks and line cards, let's take a look at the most sophisticated configurations that use all available features.

MPC1 and MPC2 with Enhanced Queuing

The only difference between the MPC1 and MPC2 at a high level is the number of Trio chipsets. Otherwise, they are operationally equivalent. Let's take a look at how a packet moves through the Trio chipset. There are couple of scenarios: ingress and egress.

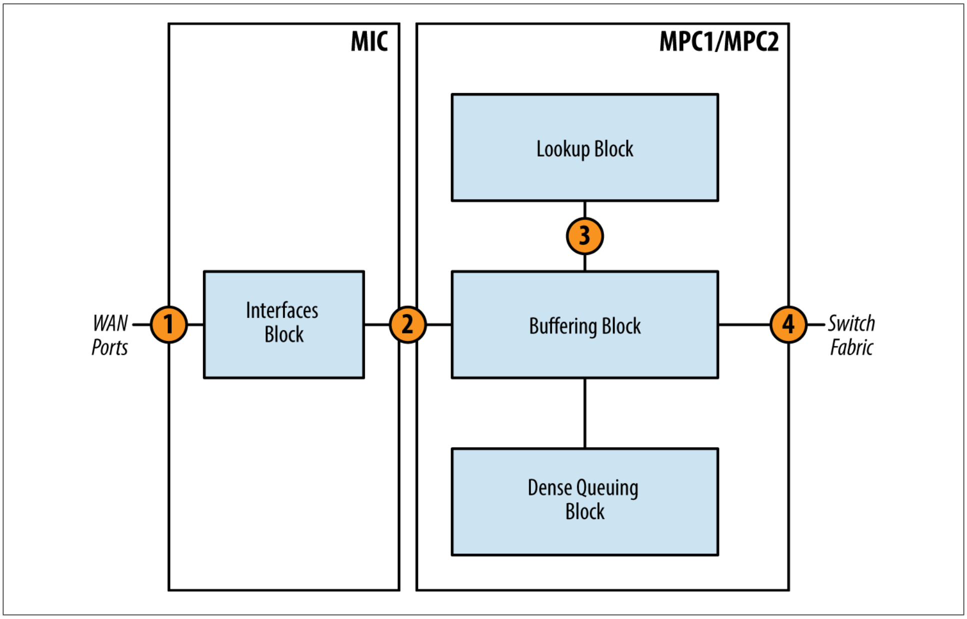


Figure 1-28. MPC1/MPC2 packet walk through: Ingress

Ingress packets are received from the WAN ports on the MIC and are destined to another PFE.

1. The packet enters the Interfaces Block from the WAN ports. The Interfaces Block will inspect each packet and perform preclassification. Depending on the type of packet, it will be marked as high or low priority.
2. The packet enters the Buffering Block. The Buffering Block will enqueue the packet as determined by the preclassification and service the high priority queue first.

3. The packet enters the Lookup Block. A route lookup is performed and any services such as firewall filters, policing, statistics, and QoS classification are performed.
4. The packet is sent back to the Buffering Block and is enqueued into the switch fabric where it will be destined to another PFE. If the packet is destined to a WAN port within itself, it will simply be enqueued back to the Interfaces Block.

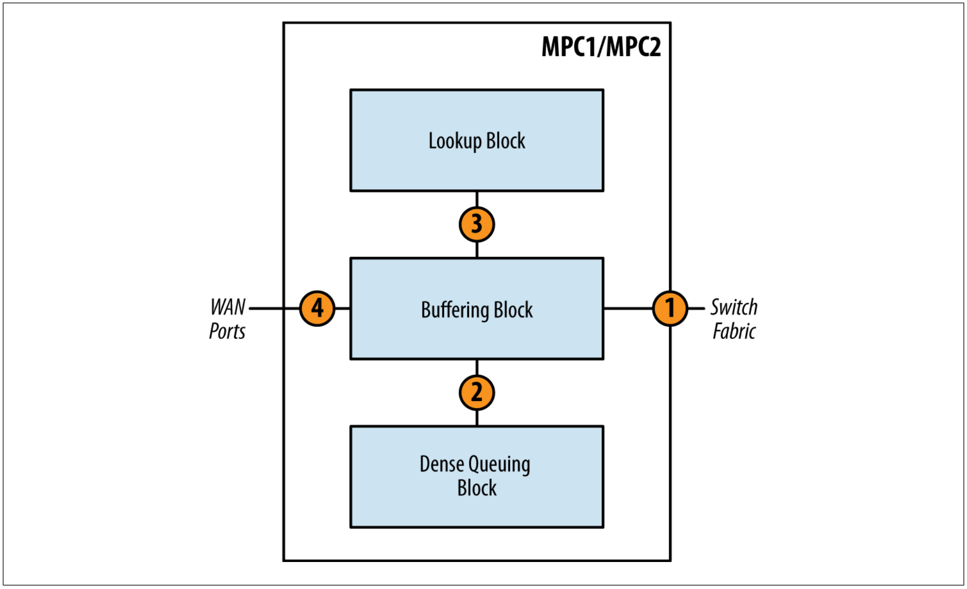


Figure 1-29. MPC1/MPC2 packet walk through: Egress

Egress packets are handled a bit differently. The major difference is that the Dense Queuing Block will perform class of service, if configured, on egress packets.

1. The packet enters the Buffering Block. If class of service is configured, the Buffering Block will send the packet to the Dense Queuing Block.
2. The packet enters the Dense Queuing Block. The packet will then be subject to scheduling, shaping, and any other hierarchical class of service as required. Packets will be enqueued as determined by the class of service configuration. The Dense Queuing Block will then dequeue packets that are ready for transmission and send them to the Buffering Block.
3. The Buffering Block receives the packet and sends it to the Lookup Block. A route lookup is performed as well as any services such as firewall filters, policing, statistics, and accounting.
4. The packet is then sent out to the WAN interfaces for transmission.

MPC3E

The packet flow of the MPC3E is similar to the MPC1 and MPC2, with a couple of notable differences: introduction of the Fabric Block and multiple Lookup Blocks. Let's review the ingress packet first:

1. The packet enters the Buffering Block from the WAN ports and is subject to pre-classification. Depending on the type of packet, it will be marked as high or low priority. The Buffering Block will enqueue the packet as determined by the pre-classification at service the high-priority queue first. A Lookup Block is selected via round-robin and the packet is sent to that particular Lookup Block.
2. The packet enters the Lookup Block. A route lookup is performed and any services such as firewall filters, policing, statistics, and QoS classification are performed. The Lookup Block sends the packet back to the Buffering Block.
3. The packet is sent back to the Fabric Block and is enqueued into the switch fabric where it will be destined to another PFE. If the packet is destined to a WAN port within itself, it will simply be enqueued back to the Interfaces Block.
4. The packet is sent to the switch fabric.

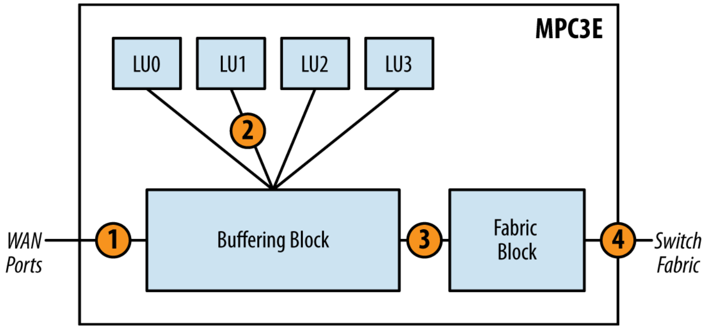


Figure 1-30. MPC3E packet walk through: Ingress.

Egress packets are very similar to ingress, but the direction is simply reversed. The only major difference is that the Buffering Block will perform basic class of service, as it doesn't support enhanced queuing due to the lack of a Dense Queuing Block.

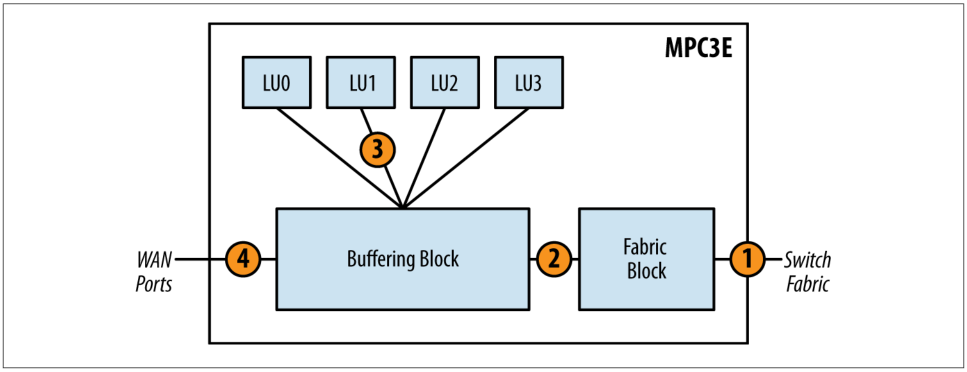


Figure 1-31. MPC3E packet Walkthrough: Egress.

1. The packet is received from the switch fabric and sent to the Fabric Block. The Fabric Block sends the packet to the Buffering Block.
2. The packet enters the Buffering Block. The packet will then be subject to scheduling, shaping, and any other class os service as required. Packets will be enqueued as determined by the class of service configuration. The Buffering Block will then dequeue packets that are ready for transmission and send them to a Lookup Block selected via round-robin.
3. The packet enters the Lookup Block. A route lookup is performed as well as any services such as firewall filters, policing, statistics, and QoS classification. The Lookup Block sends the packet back to the Buffering Block.
4. The Buffering Block receives the packet and sends it to the WAN ports for transmission.

Modular Interface Card

As described previously, the MICs provide the physical ports and are modules that are to be installed into various MPCs. Two MICs can be installed into any of the MPCs. There is a wide variety of physical port configurations available. The speeds range from 1G to 100G and support different media such as copper or optical.

MIC-3D-20GE-SFP
  Supports 20x1G SFP ports

MIC-3D-40GE-TX
  Double-wide MIC that supports 40x1G RJ-45

MIC-3D-2XGE-XFP
  Supports 2x10G XFP ports

MIC-3D-4XGE-XFP
  Supports 4x10G XFP ports; only operates in UPOH mode

MIC-3D-1X100G-CFP
: Supports 1x100G CFP port

MIC-3D-4CHOC3-2CHOC12
: Supports four ports of channelized OC-3 or two ports of channelized OC-12

MIC-3D-4OC4OC12-1OC48
: Supports four ports of nonchannelized OC-3 or OC-12 or one port of nonchannelized OC-48

MIC-3D-8CHOC3-4CHOC12
: Supports eight ports of channelized OC-3 or four ports of channelized OC-12

MIC-3D-8OC3OC12-4OC48
: Supports eight ports of nonchannelized OC-3 through OC-12 or four ports of nonchannelized OC-48

MIC-3D-8CHDS3-E3-B
: Supports eight ports of channelized DS3 or non-channelized E3

MIC-3D-8DS3-E3
: Supports eight ports of nonchannelized DS3 or nonchannelized E3

> The MIC-3D-40GE-TX is a bit of an odd man out as it's a double-wide MIC that consumes both MIC slots on the MPC.

Being modular in nature, the MICs are able to be moved from one MPC to another. They are hot-swappable and do not require a reboot to take effect. MICs offer the greatest investment protection as they're able to be used across all of the MX platforms and various MPCs. However, there are a few caveats specific to the 4x10GE and 1x100GE MICs. Please see the following compatibility table to determine what MIC can be used where.

Table 1-9. MIC compatibility chart

MIC	MPC1	MPC2	MPC3	MX80	MX240	MX480	MX960
MIC-3D-20GE-SFP	Yes	Yes	Yes	Yes	Yes	Yes	Yes
MIC-3D-40GE-TX	Yes	Yes	*No*	Yes	Yes	Yes	Yes
MIC-3D-2XGE-XFP	Yes	Yes	Yes	Yes	Yes	Yes	Yes
MIC-3D-4XGE-XFP	*No*	Yes	*No*	*No*	Yes	Yes	Yes
MIC-3D-1X100G-CFP	*No*	*No*	Yes	*No*	Yes	Yes	Yes
MIC-3D-4CHOC3-2CHOC12	Yes	Yes	*No*	Yes	Yes	Yes	Yes
MIC-3D-4OC3OC12-1OC48	Yes	Yes	*No*	Yes	Yes	Yes	Yes
MIC-3D-8CHOC3-4CHOC12	Yes	Yes	*No*	Yes	Yes	Yes	Yes
MIC-3D-8OC3OC12-4OC48	Yes	Yes	*No*	Yes	Yes	Yes	Yes

MIC	MPC1	MPC2	MPC3	MX80	MX240	MX480	MX960
MIC-3D-8CHDS3-E3-B	Yes	Yes	*No*	Yes	Yes	Yes	Yes
MIC-3D-8DS3-E3	Yes	Yes	*No*	Yes	Yes	Yes	Yes

Network Services

The MX240, MX480, and MX960 are able to operate with different types of line cards at the same time. For example, it's possible to have a MX240 operate with FPC1 using a DPCE-R line card while FPC2 using a MX-MPC-R-B line card. Because there are many different variations of DPC, MPC, Ethernet, and routing options, a chassis control feature called network services can be used force the chassis into a particular compatibility mode.

If the network services aren't configured, then by default when a MX chassis boots up, the FPC that is powered up first will determine the mode of the chassis. If the first FPC to be powered up is DPC, then only DPCs within the chassis will be allowed to power up. Alternatively, if the first powered up FPC is MPC, then only MPCs within the chassis will be allowed to power up.

The chassis network services can be configured with `set chassis network-services` knob. There are five different options the network services can be set to:

`ip`
: Allow all line cards to power up, except for DPCE-X. The `ip` hints toward being able to route, thus line cards such as the DPCE-X will not be allowed to power up as they only support bridging.

`ethernet`
: Allow all line cards to power up. This includes the DPCE-X, DPCE-R, and DPCE-Q.

`enhanced-ip`
: Allow all Trio-based MPCs to be powered up.

`enhanced-ethernet`
: Allow only Trio-based MPC-3D, MPC-3D-Q, and MPC-3D-EQ line cards to be powered up.

`all-ip`
: Allow both DPC and MPC line cards to be powered up, except for DPCE-X line cards. This option was hidden in Junos 10.0 and was used for manufacturing testing.

`all-ethernet`
: Allow both DPC and MPC line cards to be powered up. This includes the DPCE-X and other line cards that are Layer 2 only. This option was hidden in Junos 10.0 and was used for manufacturing testing.

The `all-ip` and `all-ethernet` modes are deprecated and shouldn't be used. These options were used exclusively for developer and manufacturing testing.

It's possible to change the value of `network services` while the chassis is running. There are many different combinations; some require a reboot, while others do not:

Change from `ip` *to* `ethernet`
: Any DPCE-X will boot up. No reboot required.

Change from `ethernet` *to* `ip`
: This change will generate a commit error. It's required that any DPCE-X line cards be powered off before the change can take effect.

Change `enhanced-ip` *to* `enhanced-ethernet`
: Any MPC-3D, MPC-3D-Q, and MPC-3D-EQ line cards will boot up. No reboot required.

Change `enhanced-ethernet` *to* `enhanced-ip`
: No change.

Change between `ip` *or* `ethernet` *to* `enhanced-ip` *or* `enhanced-ethernet`
: The commit will complete but will require a reboot of the chassis.

To view which mode the network services is currently set to, use the `show chassis network-services` command:

```
dhanks@R1> show chassis network-services
Network Services Mode: IP

dhanks@R1>
```

Switch and Control Board

At the heart of the MX Series is the Switch and Control Board (SCB). It's the glue that brings everything together. The SCB has three primary functions: switch data between the line cards, control the chassis, and house the routing engine. The SCB is a single-slot card and has a carrier for the routing engine on the front. A SCB contains the following components:

- An Ethernet switch for chassis management
- Two switch fabrics
- Control board (CB) and routing engine state machine for mastership arbitration
- Routing engine carrier

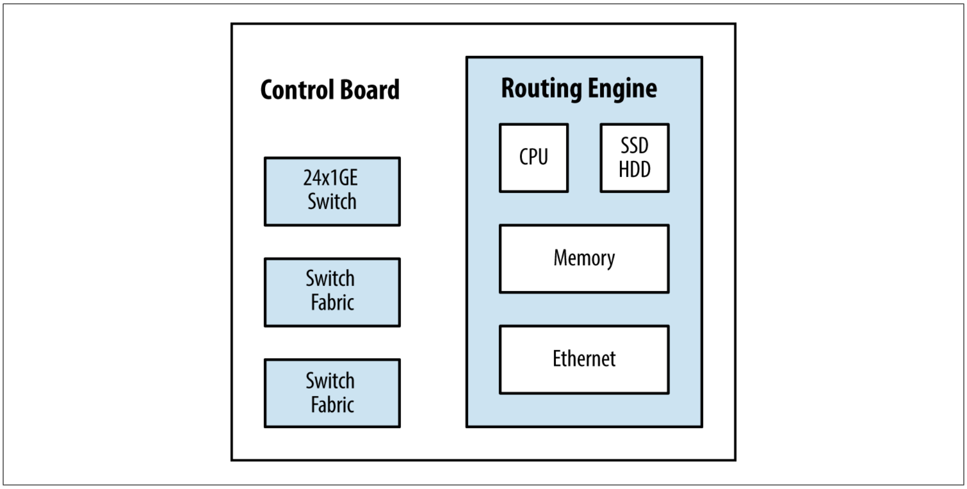


Figure 1-32. Switch and control board components

Depending on the chassis and level of redundancy, the number of SCBs vary. The MX240 and MX480 require two SCBs for 1 + 1 redundancy, whereas the MX960 requires three SCBs for 2 + 1 redundancy.

Ethernet Switch

Each SCB contains a 24-port Gigabit Ethernet switch. This internal switch connects the two routing engines and all of the FPCs together. Each routing engine has two networking cards. The first NIC is connected to the local onboard Ethernet switch, whereas the second NIC is connected to the onboard Ethernet switch on the other SCB. This allows the two routing engines to have internal communication for features such as NSR, NSB, ISSU, and administrative functions such as copying files between the routing engines.

Each Ethernet switch has connectivity to each of the FPCs. This allows for the routing engines to communicate to the Junos microkernel onboard each of the FPCs. A good example would be when a packet needs to be processed by the routing engine. The FPC would need to send the packet across the SCB Ethernet switch and up to the master routing engine. Another good example is when the routing engine modifies the forwarding information base (FIB) and updates all of the PFEs with the new information.

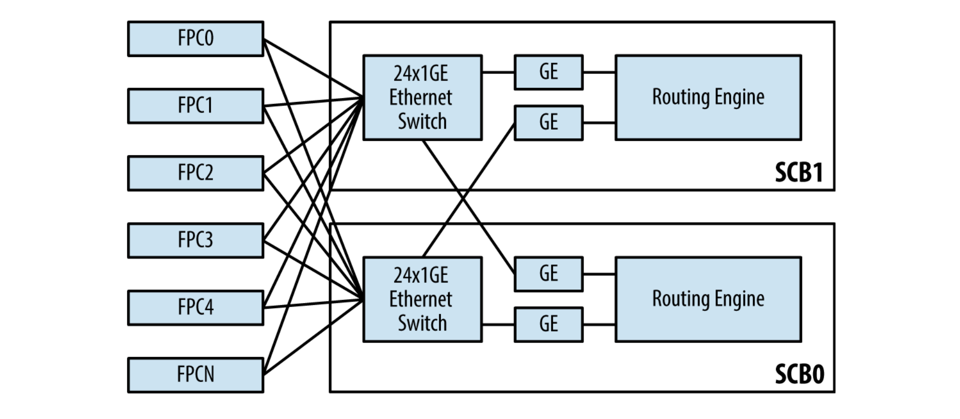


Figure 1-33. MX-SCB Ethernet switch connectivity

It's possible to view information about the Ethernet switch inside of the SCB. The command `show chassis ethernet-switch` will show which ports on the Ethernet switch are connected to which devices at a high level.

```
{master}
dhanks@R1-RE0> show chassis ethernet-switch

Displaying summary for switch 0
Link is good on GE port 1 connected to device: FPC1
  Speed is 1000Mb
  Duplex is full
  Autonegotiate is Enabled
  Flow Control TX is Disabled
  Flow Control RX is Disabled

Link is good on GE port 2 connected to device: FPC2
  Speed is 1000Mb
  Duplex is full
  Autonegotiate is Enabled
  Flow Control TX is Disabled
  Flow Control RX is Disabled

Link is good on GE port 12 connected to device: Other RE
  Speed is 1000Mb
  Duplex is full
  Autonegotiate is Enabled
  Flow Control TX is Disabled
  Flow Control RX is Disabled

Link is good on GE port 13 connected to device: RE-GigE
  Speed is 1000Mb
  Duplex is full
  Autonegotiate is Enabled
  Flow Control TX is Disabled
  Flow Control RX is Disabled
  Receive error count = 012032
```

The Ethernet switch will only be connected to FPCs that are online and routing engines. As you can see, R1-RE0 is showing that its Ethernet switch is connected to both FPC1 and FPC2. Let's check the hardware inventory to make sure that this information is correct.

```
{master}
dhanks@R1-RE0> show chassis fpc
                     Temp  CPU Utilization (%)   Memory    Utilization (%)
Slot State            (C)  Total  Interrupt      DRAM (MB) Heap     Buffer
  0  Empty
  1  Online            35     21          0       2048       12         13
  2  Online            34     22          0       2048       11         16

{master}
dhanks@R1-RE0>
```

As you can see, FPC1 and FPC2 are both online. This matches the previous output from the `show chassis ethernet-switch`. Perhaps the astute reader noticed that the Ethernet switch port number is paired with the FPC location. For example, GE port 1 is connected to FPC1 and GE port 2 is connected to FPC2, so on and so forth all the way up to FPC11.

Although each Ethernet switch has 24 ports, only 14 are being used. GE ports 0 through 11 are reserved for FPCs, while GE ports 12 and 13 are reserved for connections to the routing engines.

Table 1-10. MX-SCB Ethernet switch port assignments

GE Port	Description
0	FPC0
1	FPC1
2	FPC2
3	FPC3
4	FPC4
5	FPC5
6	FPC6
7	FPC7
8	FPC8
9	FPC9
10	FPC10
11	FPC11
12	Other Routing Engine
13	Routing Engine GE

One interesting note is that the `show chassis ethernet-switch` command is relative to where it's executed. GE port 12 will always be the other routing engine. For example, when the command is executed from re0, the GE port 12 would be connected to re1 and GE port 13 would be connected to re0.

To view more detailed information about a particular GE port on the SCB Ethernet switch, you can use the command `show chassis ethernet-switch` statistics command. Let's take a closer look at GE port 13, which is connected to the local routing engine.

```
{master}
dhanks@R1-RE0> show chassis ethernet-switch statistics 13

Displaying port statistics for switch 0
Statistics for port 13 connected to device RE-GigE:
  TX Packets 64 Octets        29023890
  TX Packets 65-127 Octets    101202929
  TX Packets 128-255 Octets   14534399
  TX Packets 256-511 Octets   239283
  TX Packets 512-1023 Octets  610582
  TX Packets 1024-1518 Octets  1191196
  TX Packets 1519-2047 Octets  0
  TX Packets 2048-4095 Octets  0
  TX Packets 4096-9216 Octets  0
  TX 1519-1522 Good Vlan frms  0
  TX Octets                   146802279
  TX Multicast Packets        4
  TX Broadcast Packets        7676958
  TX Single Collision frames  0
  TX Mult. Collision frames   0
  TX Late Collisions          0
  TX Excessive Collisions     0
  TX Collision frames         0
  TX PAUSEMAC Ctrl Frames     0
  TX MAC ctrl frames          0
  TX Frame deferred Xmns      0
  TX Frame excessive deferl   0
  TX Oversize Packets         0
  TX Jabbers                  0
  TX FCS Error Counter        0
  TX Fragment Counter         0
  TX Byte Counter             2858539809

<output truncated for brevity>
```

Although the majority of the traffic is communication between the two routing engines, exception traffic is also passed through the Ethernet switch. When an ingress PFE receives a packet that needs additional processing—such as a BGP update or SSH traffic destined to the router—the packet needs to be encapsulated and sent to the routing engine. The same is true if the routing engine is sourcing traffic that needs to be sent out an egress PFE.

Switch Fabric

The switch fabric connects all of the ingress and egress PFEs within the chassis to create a full mesh. Each SCB has two switch fabrics. Depending on the MX chassis, each switch fabric can have either one or two fabric planes.

The MX240 and MX480 support two SCBs for a total of four switch fabrics and eight fabric planes. The MX960 supports three SCBs for a total of six switch fabrics and six fabric planes.

This begs the question, what is a fabric plane? Think of the switch fabric as a fixed unit that can support N connections. When supporting 48 PFEs on the MX960, all of these connections on the switch fabric are completely consumed. Now think about what happens when you apply the same logic to the MX480. Each switch fabric now only has to support 24 PFEs, thus half of the connections aren't being used. What happens on the MX240 and MX480 is that these unused connections are grouped together and another plane is created so that the unused connections can now be used. The benefit is that the MX240 and MX480 only require a single SCB to provide line rate throughput, thus only require an additional SCB for 1 + 1 SCB redundancy.

Table 1-11. MX-SCB fabric plane scale and redundancy assuming four PFEs per FPC

MX-SCB	MX240	MX480	MX960
PFEs	12	24	48
SCBs	2	2	3
Switch Fabrics	4	4	6
Fabric Planes	8	8	6
Spare Planes	4 (1 + 1 SCB redundancy)	4 (1 + 1 SCB redundancy)	2 (2 + 1 SCB redundancy)

MX240 and MX480 Fabric Planes

Given that the MX240 and MX480 only have to support a fraction of the number of PFEs as the MX960, we're able to group together the unused connections on the switch fabric and create a second fabric plane per switch fabric. Thus we're able to have two fabric planes per switch fabric, as shown in Figure 1-34.

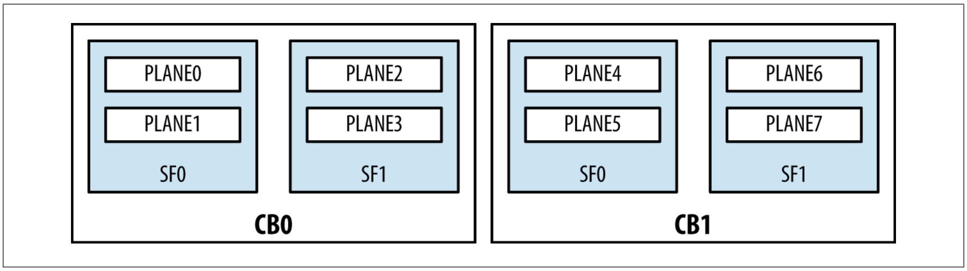


Figure 1-34. Juniper MX240 and MX480 switch fabric planes

As you can see, each control board has two switch fabrics: SF0 and SF1. Each switch fabric has two fabric planes. Thus the MX240 and MX480 have eight available fabric planes. This can be verified with the command `show chassis fabric plane-location`.

```
{master}
dhanks@R1-RE0> show chassis fabric plane-location
------------Fabric Plane Locations-------------
Plane 0                         Control Board 0
Plane 1                         Control Board 0
Plane 2                         Control Board 0
Plane 3                         Control Board 0
Plane 4                         Control Board 1
Plane 5                         Control Board 1
Plane 6                         Control Board 1
Plane 7                         Control Board 1

{master}
dhanks@R1-RE0>
```

Because the MX240 and MX480 only support two SCBs, they support 1 + 1 SCB redundancy. By default, SCB0 is in the *Online* state and processes all of the forwarding. SCB1 is in the *Spare* state and waits to take over in the event of a SCB failure. This can be illustrated with the command `show chassis fabric summary`.

```
{master}
dhanks@R1-RE0> show chassis fabric summary
Plane   State    Uptime
 0      Online   18 hours, 24 minutes, 57 seconds
 1      Online   18 hours, 24 minutes, 52 seconds
 2      Online   18 hours, 24 minutes, 51 seconds
 3      Online   18 hours, 24 minutes, 46 seconds
 4      Spare    18 hours, 24 minutes, 46 seconds
 5      Spare    18 hours, 24 minutes, 41 seconds
 6      Spare    18 hours, 24 minutes, 41 seconds
 7      Spare    18 hours, 24 minutes, 36 seconds

{master}
dhanks@R1-RE0>
```

As expected, planes 0 to 3 are `Online` and planes 4 to 7 are `Spare`. Another useful tool from this command is the *Uptime*. The *Uptime* column displays how long the SCB has been up since the last boot. Typically, each SCB will have the same uptime as the system itself, but it's possible to hot-swap SCBs during a maintenance; the new SCB would then show a smaller uptime than the others.

MX960 Fabric Planes

The MX960 is a different beast because of the PFE scale involved. It has to support twice the number of PFEs as the MX480, while maintaining the same line rate performance requirements. An additional SCB is mandatory to support these new scaling and performance requirements.

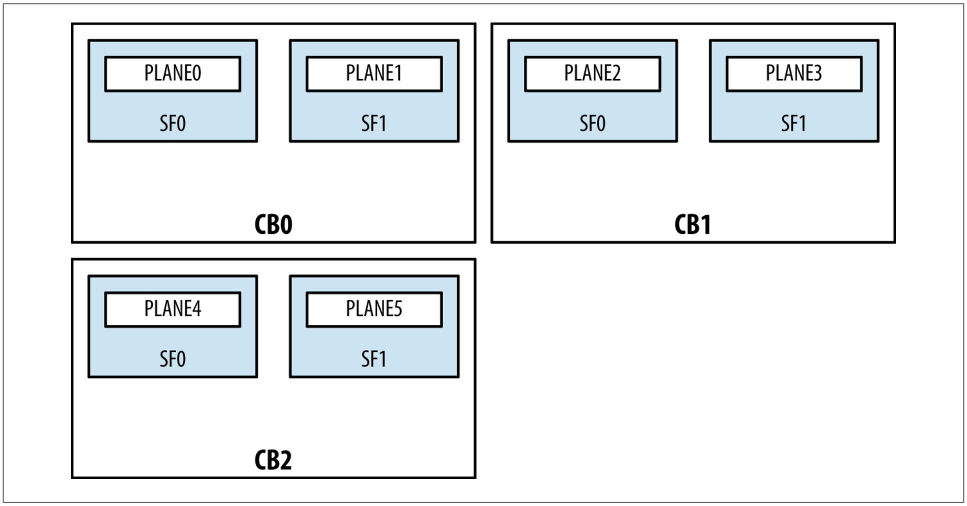


Figure 1-35. Juniper MX960 switch fabric planes

Unlike the MX240 and MX480, the switch fabrics only support a single fabric plane because all available links are required to create a full mesh between all 48 PFEs. Let's verify this with the command `show chassis fabric plane-location`.

```
{master}
dhanks@MX960> show chassis fabric plane-location
------------Fabric Plane Locations-------------
Plane 0                        Control Board 0
Plane 1                        Control Board 0
Plane 2                        Control Board 1
Plane 3                        Control Board 1
Plane 4                        Control Board 2
Plane 5                        Control Board 2

{master}
dhanks@MX960>
```

As expected, things seem to line up nicely. We see there are two switch fabrics per control board. The MX960 supports up to three SCBs providing 2 + 1 SCB redundancy. At least two SCBs are required for basic line rate forwarding, and the third SCB provides redundancy in case of a SCB failure. Let's take a look at the command `show chassis fabric summary`.

```
{master}
dhanks@MX960> show chassis fabric summary
Plane   State    Uptime
 0      Online   18 hours, 24 minutes, 22 seconds
 1      Online   18 hours, 24 minutes, 17 seconds
 2      Online   18 hours, 24 minutes, 12 seconds
 3      Online   18 hours, 24 minutes, 6 seconds
 4      Spare    18 hours, 24 minutes, 1 second
 5      Spare    18 hours, 23 minutes, 56 seconds
```

```
{master}
dhanks@MX960>
```

Everything looks good. SCB0 and SCB1 are *Online*, whereas the redundant SCB2 is standing by in the `Spare` state. If SCB0 or SCB1 fails, SCB2 will immediately transition to the `Online` state and allow the router to keep forwarding traffic at line rate.

J-Cell

As packets move through the MX from one PFE to another, they need to traverse the switch fabric. Before the packet can be placed onto the switch fabric, it first must be broken into J-cells. A J-cell is a 64-byte fixed-width unit.

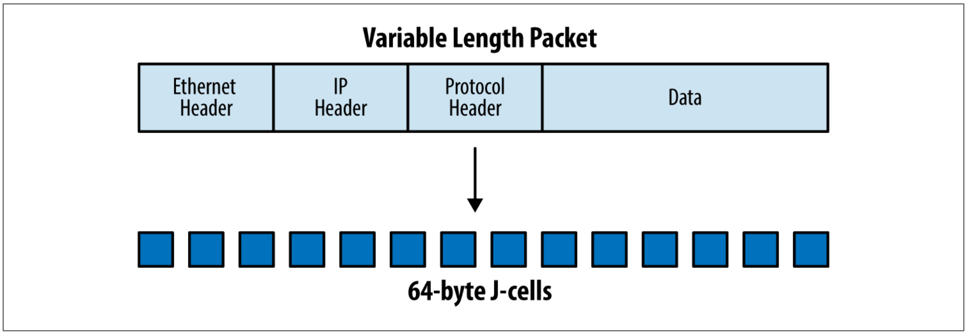


Figure 1-36. Cellification of variable length packets

The benefit of J-cells is that it's much easier for the router to process, buffer, and transmit fixed-width data. When dealing with variable length packets with different types of headers, it adds inconsistency to the memory management, buffer slots, and transmission times. The only drawback when segmenting variable data into a fixed-width unit is the waste, referred to as "cell tax." For example, if the router needed to segment a 65-byte packet, it would require two J-cells: the first J-cell would be fully utilized, the second J-cell would only carry 1 byte, and the other 63 bytes of the J-cell would go unused.

For those of you old enough (or savvy enough) to remember ATM, go ahead and laugh.

J-Cell Format

There are some additional fields in the J-cell to optimize the transmission and processing:

- Request source and destination address
- Grant source and destination address
- Cell type
- Sequence number
- Data (64 bytes)
- Checksum

Each PFE has an address that is used to uniquely identify it within the fabric. When J-cells are transmitted across the fabric a source and destination address is required, much like the IP protocol. The sequence number and cell type aren't used by the fabric, but instead are important only to the destination PFE. The sequence number is used by the destination PFE to reassemble packets in the correct order. The cell type identifies the cell as one of the following: first, middle, last, or single cell. This information assists in the reassembly and processing of the cell on the destination PFE.

J-Cell Flow

As the packet leaves the ingress PFE, the Trio chipset will segment the packet into J-cells. Each J-cell will be sprayed across all available fabric links. The following illustration represents a MX960 fully loaded with 48 PFEs and 3 SCBs. The example packet flow is from left to right.

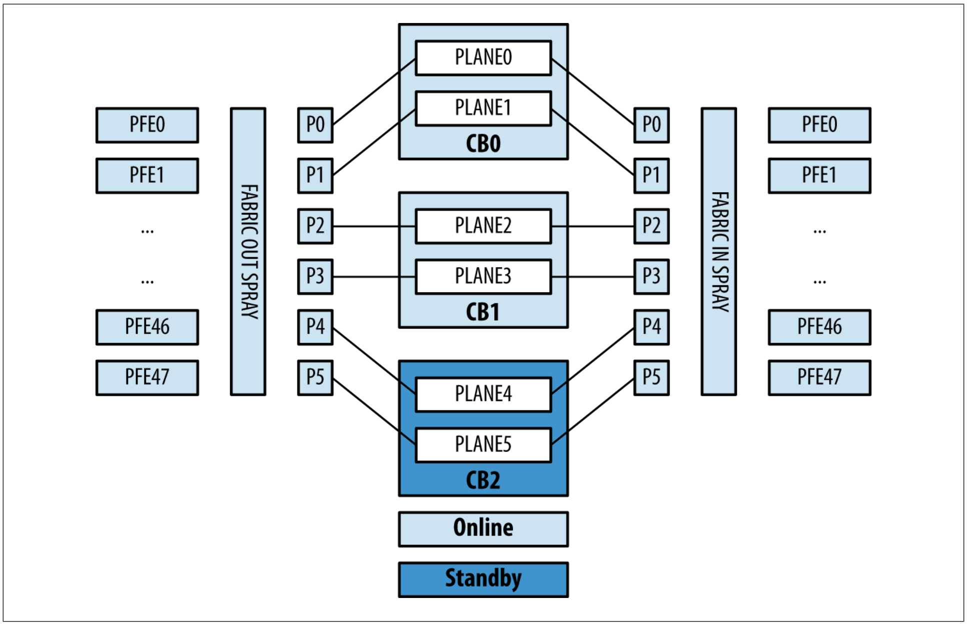


Figure 1-37. Juniper MX960 fabric spray and reordering across the MX-SCB

J-cells will be sprayed across all available fabric links. Keep in mind that only `PLANE0` through `PLANE3` are `Online`, whereas `PLANE4` and `PLANE5` are `Standby`.

Request and Grant

Before the J-cell can be transmitted to the destination PFE, it needs to go through a three-step request and grant process:

1. The source PFE will send a request to the destination PFE.
2. The destination PFE will respond back to the source PFE with a grant.
3. The source PFE will transmit the J-cell.

The request and grant process guarantees the delivery of the J-cell through the switch fabric. An added benefit of this mechanism is the ability to quickly discover broken paths within the fabric and provide a method of flow control.

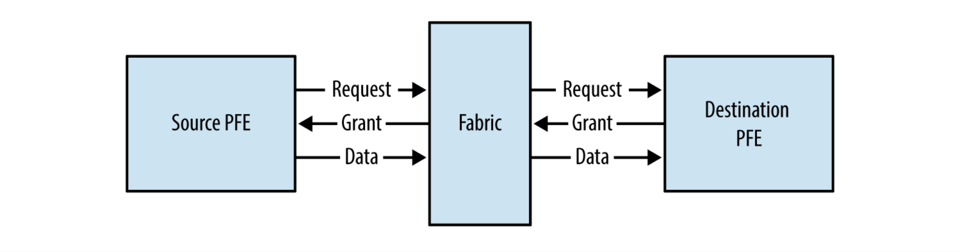


Figure 1-38. Juniper MX fabric request and grant process

As the J-cell is placed into the switch fabric, it's placed into one of two fabric queues: high or low. In the scenario where there are multiple source PFEs trying to send data to a single destination PFE, it's going to cause the destination PFE to be oversubscribed. One tool that's exposed to the network operator is the fabric priority knob in the class of service configuration. When you define a forwarding class, you're able to set the fabric priority. By setting the fabric priority to high for a specific forwarding class, it will ensure that when a destination PFE, is congested, the high-priority traffic will be delivered. This is covered more in detail in Chapter 5.

MX Switch Control Board

The MX SCB is the first-generation switch fabric for the MX240, MX480, and MX960. This MX SCB was designed to work with the first-generation DPC line cards. As described previously, the MX SCB provides line-rate performance with full redundancy.

The MX240 and MX480 provide 1 + 1 MX SCB redundancy when used with the DPC line cards. The MX960 provides 2 + 1 MX SCB redundancy when used with the DPC line cards.

Each of the fabric planes on the first-generation SCB is able to process 20 Gbps of bandwidth. The MX240 and MX480 use eight fabric planes across two SCBs, whereas the MX960 uses six fabric planes across three SCBs. Because of the fabric plane virtualization, the aggregate fabric bandwidth between the MX240, MX480, and MX960 is different.

Table 1-12. First-Generation SCB bandwidth

Model	SCBs	Switch Fabrics	Fabric Planes	Fabric Bandwidth per Slot
MX240	2	4	8	160 Gbps
MX480	2	4	8	160 Gbps
MX960	3	6	6	120 Gbps

MX SCB and MPC Caveats

The only caveat is that the first-generation MX SCBs are not able to provide line-rate redundancy with some of the new-generation MPC line cards. When the MX SCB is used with the newer MPC line cards, it places additional bandwidth requirements onto the switch fabric. The additional bandwidth requirements come at a cost of oversubscription and a loss of redundancy.

> The new-generation Enhanced MX SCB is required to provide line-rate fabric bandwidth with full redundancy for high-density MPC line cards such as the MPC-3D-16x10GE-SFPP.

MX240 and MX480

As described previously, the MX240 and MX480 have a total of eight fabric planes when using two MX SCBs. When the MX SCB and MPCs are being used on the MX240 and MX480, there's no loss in performance and all MPCs are able to operate at line rate. The only drawback is that all fabric planes are in use and are Online.

Let's take a look at a MX240 with the first-generation MX SCBs and new-generation MPC line cards.

```
{master}
dhanks@R1-RE0> show chassis hardware | match FPC
FPC 1            REV 15   750-031088   ZB7956             MPC Type 2 3D Q
FPC 2            REV 25   750-031090   YC5524             MPC Type 2 3D EQ

{master}
dhanks@R1-RE0> show chassis hardware | match SCB
CB 0             REV 03   710-021523   KH6172             MX SCB
CB 1             REV 10   710-021523   ABBM2781           MX SCB

{master}
dhanks@R1-RE0> show chassis fabric summary
Plane   State    Uptime
```

```
 0      Online   10 days, 4 hours, 47 minutes, 47 seconds
 1      Online   10 days, 4 hours, 47 minutes, 47 seconds
 2      Online   10 days, 4 hours, 47 minutes, 47 seconds
 3      Online   10 days, 4 hours, 47 minutes, 47 seconds
 4      Online   10 days, 4 hours, 47 minutes, 47 seconds
 5      Online   10 days, 4 hours, 47 minutes, 46 seconds
 6      Online   10 days, 4 hours, 47 minutes, 46 seconds
 7      Online   10 days, 4 hours, 47 minutes, 46 seconds
```

As we can see, `R1` has the first-generation MX SCBs and new-generation MPC2 line cards. In this configuration, all eight fabric planes are *Online* and processing J-cells.

If a MX SCB fails on a MX240 or MX480 using the new-generation MPC line cards, the router's performance will degrade gracefully. Losing one of the two MX SCBs would result in a loss of half of the router's performance.

MX960

In the case of the MX960, it has six fabric planes when using three MX SCBs. When the first-generation MX SCBs are used on a MX960 router, there isn't enough fabric bandwidth to provide line-rate performance for the MPC-3D-16X10GE-SPFF or MPC3-3D line cards. However, with the MPC1 and MPC2 line cards, there's enough fabric capacity to operate at line rate, \ except when used with the 4x10G MIC.

Let's take a look at a MX960 with a first-generation MX SCB and second-generation MPC line cards.

```
dhanks@MX960> show chassis hardware | match SCB
CB 0              REV 03.6 710-013385   JS9425           MX SCB
CB 1              REV 02.6 710-013385   JP1731           MX SCB
CB 2              REV 05   710-013385   JS9744           MX SCB

dhanks@MX960> show chassis hardware | match FPC
FPC 2             REV 14   750-031088   YH8454           MPC Type 2 3D Q
FPC 5             REV 29   750-031090   YZ6139           MPC Type 2 3D EQ
FPC 7             REV 29   750-031090   YR7174           MPC Type 2 3D EQ

dhanks@MX960> show chassis fabric summary
Plane   State    Uptime
 0      Online   11 hours, 21 minutes, 30 seconds
 1      Online   11 hours, 21 minutes, 29 seconds
 2      Online   11 hours, 21 minutes, 29 seconds
 3      Online   11 hours, 21 minutes, 29 seconds
 4      Online   11 hours, 21 minutes, 28 seconds
 5      Online   11 hours, 21 minutes, 28 seconds
```

As you can see, the MX960 has three of the first-generation MX SCB cards. There's also three second-generation MPC line cards. Taking a look at the fabric summary, we can surmise that all six fabric planes are `Online`. When using high-speed MPCs and MICs, the oversubscription is approximately 4:3 with the first-generation MX SCB. Losing a MX SCB with the new-generation MPC line cards would cause the MX960 to gracefully degrade performance by a third.

Enhanced MX Switch Control Board

The second-generation Enhanced MX Switch Control Board (SCBE) doubles performance from the previous MX SCB. The SBCE was designed to be used specifically with the new-generation MPC line cards to provide full line-rate performance and redundancy without a loss of bandwidth.

Table 1-13. Second-generation SCBE bandwidth

Model	SCBs	Switch Fabrics	Fabric Planes	Fabric Bandwidth Per Slot
MX240	2	4	8	320 Gbps
MX480	2	4	8	320 Gbps
MX960	3	6	6	240 Gbps

MX240 and MX480

When the SCBE is used with the MX240 and MX480, only one SCBE is required for full line-rate performance and redundancy.

Let’s take a look at a MX480 with two SCBEs and 100G MPC3 line cards.

```
dhanks@paisa> show chassis hardware | match SCB
CB 0             REV 14   750-031391   ZK8231           Enhanced MX SCB
CB 1             REV 14   750-031391   ZK8226           Enhanced MX SCB

dhanks@paisa> show chassis hardware | match FPC
FPC 0            REV 24   750-033205   ZJ6553           MPC Type 3
FPC 1            REV 21   750-033205   ZG5027           MPC Type 3

dhanks@paisa> show chassis fabric summary
Plane   State    Uptime
 0      Online   5 hours, 54 minutes, 51 seconds
 1      Online   5 hours, 54 minutes, 45 seconds
 2      Online   5 hours, 54 minutes, 45 seconds
 3      Online   5 hours, 54 minutes, 40 seconds
 4      Spare    5 hours, 54 minutes, 40 seconds
 5      Spare    5 hours, 54 minutes, 35 seconds
 6      Spare    5 hours, 54 minutes, 35 seconds
 7      Spare    5 hours, 54 minutes, 30 seconds
```

Much better. You can see that there are two SCBEs as well 100G MPC3 line cards. When taking a look at the fabric summary, we see that all eight fabric planes are present. The big difference is that now four of the planes are `Online` while the other four are `Spare`. These new SCBEs are providing line-rate fabric performance as well as 1 + 1 SCB redundancy.

Because the MX SCBE is twice the performance of the previous MX SCB, the MX960 can now go back to the original 2 + 1 SCB for full line-rate performance and redundancy.

MX960

Let’s check out a MX960 using three MX SCBEs and 100G MPC3 line cards.

```
dhanks@bellingham> show chassis hardware | match SCB
CB 0             REV 10   750-031391   ZB9999            Enhanced MX SCB
CB 1             REV 10   750-031391   ZC0007            Enhanced MX SCB
CB 2             REV 10   750-031391   ZC0001            Enhanced MX SCB

dhanks@bellingham> show chassis hardware | match FPC
FPC 0            REV 14.3.09 750-033205 YY8443           MPC Type 3
FPC 3            REV 12.3.09 750-033205 YR9438           MPC Type 3
FPC 4            REV 27   750-033205   ZL5997            MPC Type 3
FPC 5            REV 27   750-033205   ZL5968            MPC Type 3
FPC 11           REV 12.2.09 750-033205 YW7060           MPC Type 3

dhanks@bellingham> show chassis fabric summary
Plane   State    Uptime
 0      Online   6 hours, 7 minutes, 6 seconds
 1      Online   6 hours, 6 minutes, 57 seconds
 2      Online   6 hours, 6 minutes, 52 seconds
 3      Online   6 hours, 6 minutes, 46 seconds
 4      Spare    6 hours, 6 minutes, 41 seconds
 5      Spare    6 hours, 6 minutes, 36 seconds
```

What a beautiful sight. We have three MX SCBEs in addition to five 100G MPC3 line cards. As discussed previously, the MX960 has six fabric planes. We can see that four of the fabric planes are `Online`, whereas the other two are `Spare`. We now have line-rate fabric performance plus 2 + 1 MX SCBE redundancy.

MX2020

The release of this book has been timed with a new product announcement from Juniper. The MX2020 is a new router in the MX Series that’s designed to solve the 10G and 100G high port density needs of Content Service Providers (CSP), Multisystem Operators (MSO), and traditional Service Providers. At a glance, the MX2020 supports 20 line cards, 8 switch fabric boards, and 2 routing engines. The chassis takes up an entire rack and has front-to-back cooling.

Architecture

The MX2020 is a standard backplane-based system, albeit at a large scale. There are two backplanes connected together with centralized switch fabric boards (SFB). The routing engine and control board is a single unit that consumes a single slot, as illustrated in Figure 1-39 on the far left and right.

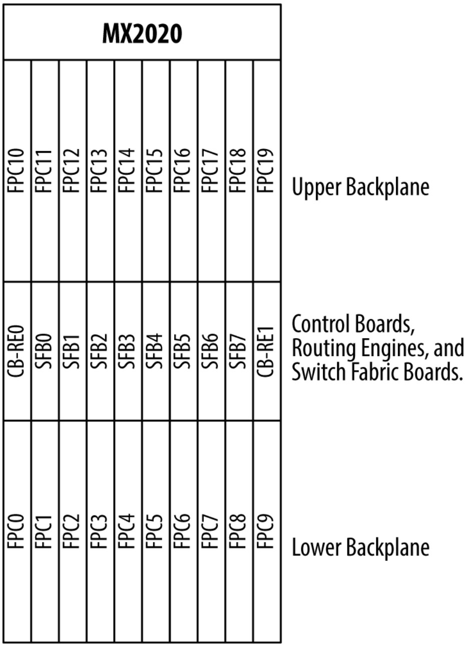


Figure 1-39. Illustration of MX2020 architecture

The FPC numbering is the standard Juniper method of starting at the bottom and moving left to right as you work your way up. The SFBs are named in the same method starting with zero starting on the left and going all the way to seven on the far right. The routing engine and control boards are located in the middle of the chassis on the far left and far right.

Switch Fabric Board

Each backplane has 10 slots that are tied into eight SFBs in the middle of the chassis. Because of the high number of line cards and PFEs the switch fabric must support, a new SFB was created specifically for the MX2020. The SFB is able to support more PFEs and has a much higher throughput compared to the previous SCBs. Recall that the SCB and SCBE presented its chipsets to Junos as a fabric plane and can be seen with the `show chassis fabric summary` command; the new SFB has multiple chipsets as well, but presents them as an aggregate fabric plane to Junos. In other words, each SFB will appear as a single fabric plane within Junos. Each SFB will be in an Active state by default. Let's take a look at the installed SFBs first:

```
dhanks@MX2020> show chassis hardware | match SFB
SFB 0            REV 01   711-032385   ZE5866         Switch Fabric Board
SFB 1            REV 01   711-032385   ZE5853         Switch Fabric Board
SFB 2            REV 01   711-032385   ZB7642         Switch Fabric Board
SFB 3            REV 01   711-032385   ZJ3555          Switch Fabric Board
SFB 4            REV 01   711-032385   ZE5850          Switch Fabric Board
SFB 5            REV 01   711-032385   ZE5870          Switch Fabric Board
SFB 6            REV 04   711-032385   ZV4182          Switch Fabric Board
SFB 7            REV 01   711-032385   ZE5858          Switch Fabric Board
```

There are eight SFBs installed; now let's take a look at the switch fabric status:

```
dhanks@MX2020> show chassis fabric summary
Plane   State    Uptime
 0      Online   1 hour, 25 minutes, 59 seconds
 1      Online   1 hour, 25 minutes, 59 seconds
 2      Online   1 hour, 25 minutes, 59 seconds
 3      Online   1 hour, 25 minutes, 59 seconds
 4      Online   1 hour, 25 minutes, 59 seconds
 5      Online   1 hour, 25 minutes, 59 seconds
 6      Online   1 hour, 25 minutes, 59 seconds
 7      Online   1 hour, 25 minutes, 59 seconds
```

Depending on which line cards are being used, only a subset of the eight SFBs need to be present in order to provide a line-rate switch fabric, but this is subject to change with line cards.

Power Supply

The power supply on the MX2020 is a bit different than the previous MX models. The MX2020 power system is split into two sections: top and bottom. The bottom power supplies provide power to the lower backplane line cards, lower fan trays, SFBs, and CB-REs. The top power supplies provide power to the upper backplane line cards and fan trays. The MX2020 provides N + 1 power supply redundancy and N + N feed redundancy. There are two major power components that supply power to the MX2020:

Power Supply Module
The Power Supply Modules (PSM) are the actual power supplies that provide power to a given backplane. There are nine PSMs per backplane, but only eight are required to fully power the backplane. Each backplane has 8 + 1 PSM redundancy.

Power Distribution Module
There are two Power Distribution Modules (PDM) per backplane, providing 1 + 1 PDM redundancy for each backplane. Each PDM contains nine PSMs to provide 8 + 1 PSM redundancy for each backplane.

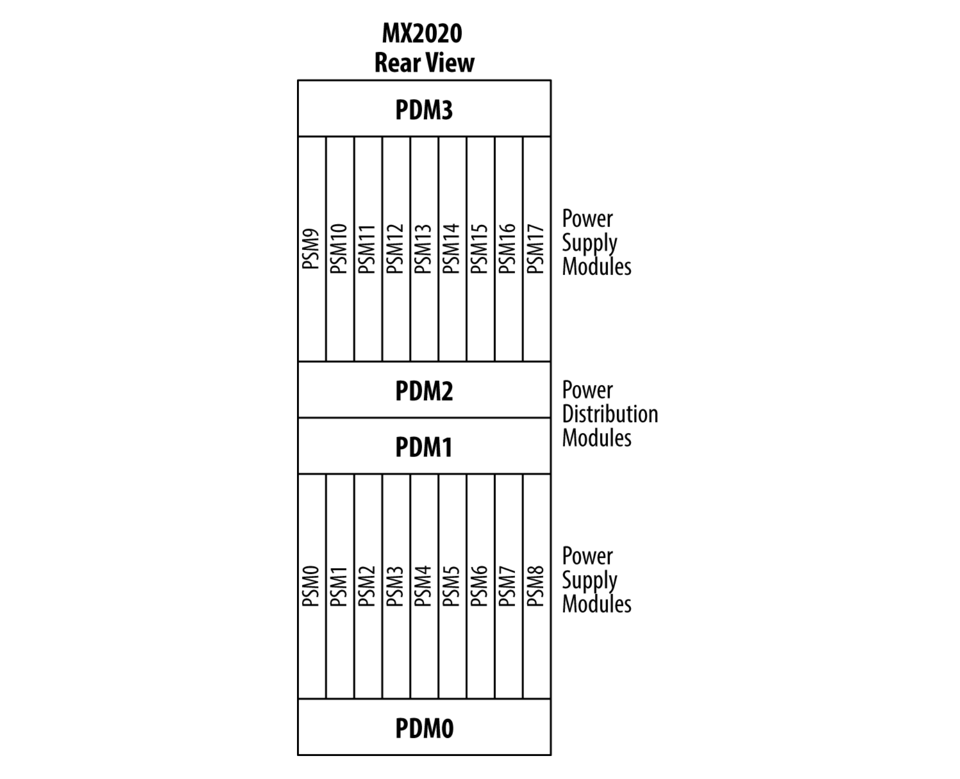


Figure 1-40. Illustration of MX2020 power supply architecture

Air Flow

The majority of data centers support hot and cold aisles, which require equipment with front to back cooling to take advantage of the airflow. The MX2020 does support front to back cooling and does so in two parts, as illustrated in Figure 1-42. The bottom inlet plenum supplies cool air from the front of the chassis and the bottom fan trays force the cool air through the bottom line cards; the air is then directed out of the back of the chassis by a diagonal airflow divider in the middle card cage. The same principal applies to the upper section. The middle inlet plenum supplies cool air from the front of the chassis and the upper fan trays push the cool air through the upper card cage; the air is then directed out the back of the chassis.

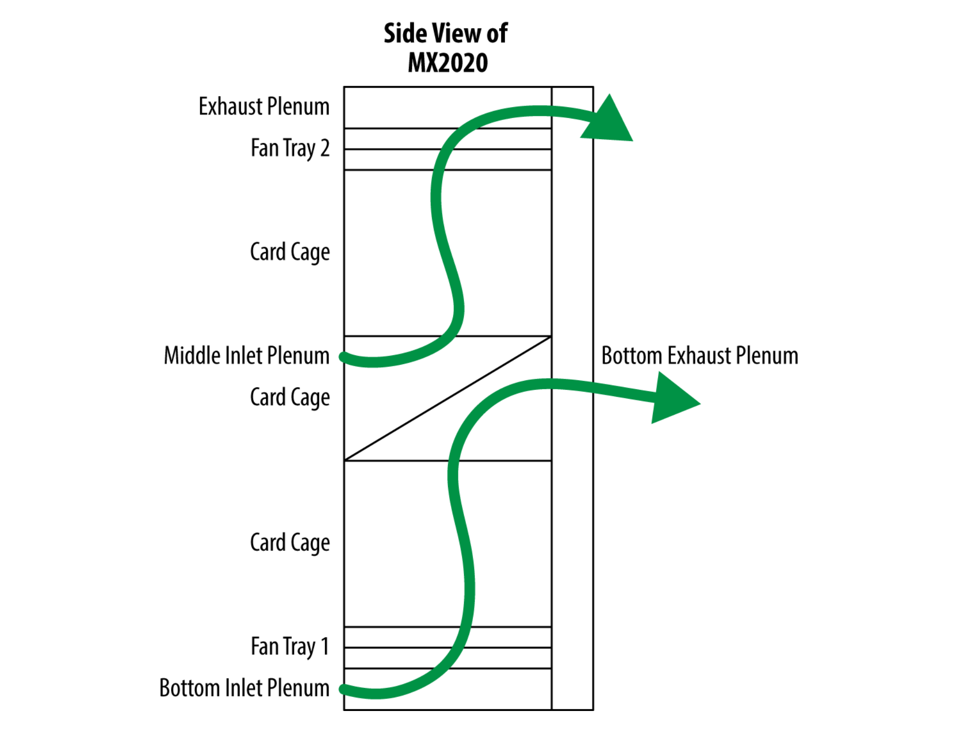


Figure 1-41. Illustration of MX2020 front-to-back air flow

Line Card Compatibility

The MX2020 is compatible with all Trio-based MPC line cards; however, there will be no backwards compatibility with the first-generation DPC line cards. The caveat is that the MPC1E, MPC2E, and MPC3E line cards will require a special MX2020 Line Card Adapter. The MX2020 can support up to 20 Adapter Cards (ADC) to accommodate 20 MPC1E through MPC3E line cards. Because the MX2020 uses a newer-generation SFB with faster bandwidth, line cards that were designed to work with the SCB and SCBE must use the ADC in the MX2020.

The ADC is merely a shell that accepts MPC1E through MPC3E line cards in the front and converts power and switch fabric in the rear. Future line cards built specifically for the MX2020 will not require the ADC. Let's take a look at the ADC status with the `show chassis adc` command:

```
dhanks@MX2020> show chassis adc
Slot  State                         Uptime
 3    Online 6 hours, 2 minutes, 52 seconds
 4    Online 6 hours, 2 minutes, 46 seconds
 8    Online 6 hours, 2 minutes, 39 seconds
 9    Online 6 hours, 2 minutes, 32 seconds
```

```
11    Online 6 hours, 2 minutes, 26 seconds
16    Online 6 hours, 2 minutes, 19 seconds
17    Online 6 hours, 2 minutes, 12 seconds
18    Online 6 hours, 2 minutes, 5 seconds
```

In this example, there are eight ADC cards in the MX2020. Let's take a closer look at FPC3 and see what type of line card is installed:

```
dhanks@MX2020> show chassis hardware | find "FPC 3"
FPC 3            REV 22   750-028467   YE2679            MPC 3D 16x 10GE
  CPU            REV 09   711-029089   YE2832            AMPC PMB
  PIC 0                   BUILTIN      BUILTIN           4x 10GE(LAN) SFP+
    Xcvr 0       REV 01   740-031980   B10M00015         SFP+-10G-SR
    Xcvr 1       REV 01   740-021308   19T511101037      SFP+-10G-SR
    Xcvr 2       REV 01   740-031980   AHK01AS           SFP+-10G-SR
  PIC 1                   BUILTIN      BUILTIN           4x 10GE(LAN) SFP+
  PIC 2                   BUILTIN      BUILTIN           4x 10GE(LAN) SFP+
    Xcvr 0       REV 01   740-021308   19T511100867      SFP+-10G-SR
  PIC 3                   BUILTIN      BUILTIN           4x 10GE(LAN) SFP+
```

The MPC-3D-16X10GE-SFPP is installed into FPC3 using the ADC for compatibility. Let's check the environmental status of the ADC installed into FPC3:

```
dhanks@MX2020> show chassis environment adc | find "ADC 3"
ADC 3 status:
  State                        Online
  Intake Temperature           34 degrees C / 93 degrees F
  Exhaust Temperature          46 degrees C / 114 degrees F
  ADC-XF1 Temperature          51 degrees C / 123 degrees F
  ADC-XF0 Temperature          61 degrees C / 141 degrees F
```

Each ADC has two chipsets, as shown in the example output: ADC-XF1 and ADC-XF2. These chipsets convert the switch fabric between the MX2020 SFB and the MPC1E through MPC3E line cards.

Aside from the simple ADC carrier to convert power and switch fabric, the MPC-3D-16X10GE-SFPP line card installed into FPC3 works just like a regular line card with no restrictions. Let's just double check the interface names to be sure:

```
dhanks@MX2020> show interfaces terse | match xe-3
Interface               Admin Link Proto    Local                 Remote
xe-3/0/0                up    down
xe-3/0/1                up    down
xe-3/0/2                up    down
xe-3/0/3                up    down
xe-3/1/0                up    down
xe-3/1/1                up    down
xe-3/1/2                up    down
xe-3/1/3                up    down
xe-3/2/0                up    down
xe-3/2/1                up    down
xe-3/2/2                up    down
xe-3/2/3                up    down
xe-3/3/0                up    down
xe-3/3/1                up    down
```

```
    xe-3/3/2                    up    down
    xe-3/3/3                    up    down
```

Just as expected: the MPC-3D-16X10GE-SFPP line card has 16 ports of 10GE interfaces grouped into four PICs with four interfaces each.

Summary

This chapter has covered a lot of topics, ranging from software to hardware. It's important to understand how the software and hardware are designed to work in conjunction with each other. This combination creates carrier-class routers that are able to solve the difficult challenges networks are facing with the explosion of high-speed and high-density Ethernet services.

Junos has a very simple and elegant design that allows for the clear and distinct separation of the control and data planes. Juniper has a principle of "distribute what you can and centralize what you must" There are a handful of functions that can be distributed to the data plane to increase performance. Examples include period packet management such as Hello packets of routing protocols and point of local repair (PLR) features such as MPLS Fast Reroute (FRR) or Loop Free Alternate (LFA) routes in routing protocols. By distributing these types of features out to the data plane, the control plane doesn't become a bottleneck and the system is to scale with ease and can restore service in under 50 ms.

The MX Series ranges from a small 2U router to a giant 44U chassis that's able to support 20 line cards. The Trio chipset is the pride and joy of the MX family; the chipset is designed for high-density and high-speed Ethernet switching and routing. Trio has the unique ability to provide inline services directly within the chipset without having to forward the traffic to a special service module. Example services include NAT, GRE, IP tunneling, port mirroring, and IP Flow Information Export (IPFIX).

The Juniper MX is such a versatile platform that it's able to span many domains and use cases. Both Enterprise Environments (EE) and Service Providers have use cases that are aligned with the feature set of the Juniper MX:

Data Center Core and Aggregation

Data centers that need to provide services to multiple tenants require multiple learning domains, routing instances, and forwarding separation. Each instance is typically mapped to a specific customer and a key requirement is collecting accounting and billing information.

Data Center Interconnect

As the number of data centers increase, the transport between them must be able to deliver the services mandated by the business. Legacy applications, storage replication, and VM mobility may require a common broadcast domain across a set of data centers. MPLS provides two methods to extend a broadcast domain across multiple sites: Virtual Private LAN Service (VPLS) and Ethernet VPN (E-VPN).

Enterprise Wide Area Network

As enterprise customers grow, the number of data centers, branch offices, and campuses increase and create a requirement to provide transport between each entity. Most customers purchase transport from a Service Provider, and the most common provider edge (PE) to customer edge (CE) routing protocol is BGP.

Service Provider Core and Aggregation

The core of a Service Provider network requires high-density and high-speed interfaces to switch MPLS labels. Features such as LFA in routing protocols and MPLS FRR are a requirement to provide PLR within 50 ms.

Service Provider Edge

The edge of Service Provider networks requires high scale in terms of routing instances, number of routing prefixes, and port density to support a large number of customers. To enforce customer service level agreements (SLA) features such as policing and hierarchical class of service (H-CoS) are required.

Broadband Subscriber Management

Multiplay and triple play services require high subscriber scale and rich features such as authentication, authorization, and accounting (AAA); change of authorization (CoA); and dynamic addressing and profiles per subscriber.

Mobile Backhaul

The number of cell phones has skyrocketed in the past 10 years and is placing high demands on the network. The varying types of service require class of service to ensure that voice calls are not queued or dropped, interactive applications are responsive, and web browsing and data transfer is best effort. Another key requirement is packet-based timing support features such as E-Sync and 1588v2.

The Juniper MX supports a wide variety of line cards that have Ethernet interfaces such as 1GE, 10GE, 40GE, and 100GE. The MPC line cards also support traditional time-division multiplexing (TDM) MICs such as T1, DS3, and OC-3. The line cards account for the bulk of the investment in the MX family, and a nice investment protection is that the line cards and MICs can be used in any Juniper MX chassis.

Each chassis is designed to provide fault protection through full hardware and software redundancy. All power supplies, fan trays, switch fabric boards, control boards, routing engines, and line cards can be host-swapped and do not require downtime to replace. Software control plane features such as graceful routing engine switchover (GRES), non-stop routing (NSR), and non-stop bridging (NSB) ensure that routing engine failures do not impact transit traffic while the backup routing engine becomes the new master. The Juniper MX chassis also supports In Service Software Upgrades (ISSU) that allows you to upgrade the software of the routing engines without impacting transit traffic or downtime. Junos high availability features will be covered in Chapter 9. The Juniper MX is a phenomenal piece of engineering that's designed from the ground up to forward packets and provide network services at all costs.

Chapter Review Questions

1. Which version of Junos is supported for three years?
    a. The first major release of the year
    b. The last maintenance release of the year
    c. The last major release of the year
    d. The last service release of the year
2. Which is not a function of the control plane?
    a. Processing SSH traffic destined to the router
    b. Updating the RIB
    c. Updating the FIB
    d. Processing a firewall filter on interface `xe-0/0/0.0`
3. How many Switch Control Boards does the MX960 require for redundancy?
    a. 1 + 1
    b. 2 + 1
    c. 1
    d. 2
4. Which is a functional block of the Trio architecture?
    a. Interfaces Block
    b. Routing Block
    c. BGP Block
    d. VLAN Block
5. Which MPC line card provides full Layer 2 and limited Layer 3 functionality?
    a. MX-3D-R-B
    b. MX-3D-Q-R-B
    c. MX-3D
    d. MX-3D-X
6. How many Trio chipsets does the MPC2 line card have?
    a. 1
    b. 2
    c. 3
    d. 4
7. What's the purpose of the Ethernet switch located on the SCB?
    a. To provide additional SCB redundancy
    b. Remote management

   c. Provide communication between line cards and routing engines
   d. To support additional H-QoS scaling

8. What J-cell attribute is used by the destination PFE to reassemble packets in the correct order?
   a. Checksum
   b. Sequence number
   c. ID number
   d. Destination address

Chapter Review Answers

1. **Answer: C**. The last major release of Junos of a given calendar year is known as the Extended End of Life (EEOL) release and is supported for three years.
2. **Answer: D**. The data/forwarding plane handles all packet processing such as firewall filters, policers, or counters on the interface `xe-0/0/0.0`.
3. **Answer: B**. The MX960 requires three SCBs for full redundancy. This is known as 2 + 1 SCB redundancy.
4. **Answer: A**. The major functional blocks of Trio are Interfaces, Buffering, Dense Queuing, and Lookup.
5. **Answer: C**. The MX-3D provides full Layer 2 and limited Layer 3 functionality. There's a limit of 32,000 prefixes in the route table.
6. **Answer: B**. The MPC2 line card has two Trio chipsets. This allows each MIC to have a dedicated Trio chipset.
7. **Answer: C**. The Ethernet switch located on the MX SCB is used to create a full mesh between all line cards and routing engines. This network processes updates and exception packets.
8. **Answer: B**. The sequence number is used to reassemble out of order packets on the destination PFE.

CHAPTER 2

Bridging, VLAN Mapping, IRB, and Virtual Switches

This chapter covers the bridging, VLAN mapping, Integrated Routing and Bridging (IRB), and virtual switch features of the Juniper MX. As you make your way through this chapter, feel free to pause and reflect on the differences between traditional bridging and advanced bridging, and where this could solve some interesting challenges in your network. Many readers may not be familiar with advanced bridging, and we encourage you to read this chapter several times. Throughout this chapter, you'll find that features such as bridge domains, learning domains, and VLAN mapping are tightly integrated, and it may be a bit challenging to follow the first time through; however, as you reread the chapter a second time, many features and caveats will become clear.

Isn't the MX a Router?

At first it may seem odd—a router is able to switch—but on the other hand it's quite common for a switch to be able to route. So what's the difference between a switch that's able to route and a router that's able to switch? Is this merely a philosophical discussion, or is there something more to it?

Traditionally, switches are designed to handle only a single Layer 2 network. A Layer 2 network is simply just a collection of one or more broadcast domains. Within the constraint of a single Layer 2 network, a switch makes sense. It's able to flood, filter, and forward traffic for 4,094 VLANs without a problem.

The problem becomes interesting as the network requirements grow, such as having to provide Ethernet services to multiple Layer 2 networks. Let's take the scenario to the next level and think about adding multiple Layer 3 networks so the requirement is to support multiple Layer 2 and Layer 3 networks on the same physical interface that has overlapping VLAN IDs, MAC addresses, and IP addresses. This challenge becomes even more interesting as you think about how to move data between these different Layer 2 and Layer 3 networks.

There's no distinction between the terms "bridging" and "switching," and they are used interchangeably in this book.

It's always helpful to see an illustration, so take a moment with Figure 2-1.

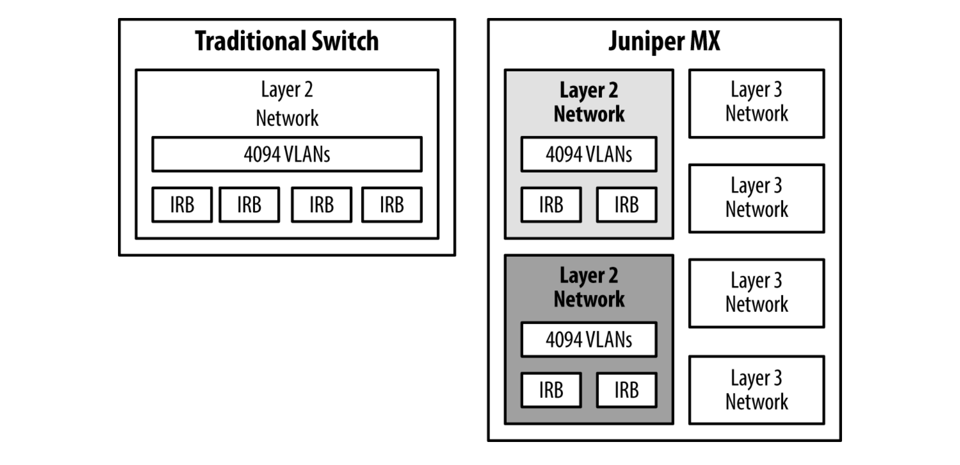


Figure 2-1. Traditional switch compared to the Juniper MX

On the left of Figure 2-1 is a traditional switch that simply supports a single Layer 2 network; within this Layer 2 network is support for 4,094 VLAN IDs and some IRB interfaces. To the right is the Juniper MX. It takes the concept of a traditional switch and virtualizes it to support multiple Layer 2 networks. This provides the ability to provide service to multiple customers with overlapping VLAN IDs.

For example, customer Green could be assigned to the upper Layer 2 network in the illustration, while customer Orange could be assigned to the lower Layer 2 network. Both customers could use identical VLAN IDs and MAC addresses without any issues using this architecture. To make it more interesting, there could be four additional customers requiring Layer 3 network services. Each customer could have overlapping IP addresses and wouldn't cause an issue.

Because of the level of virtualization, each customer is unaware of other customers within the Juniper MX. This virtualization is performed through the use of what Junos calls a *routing instance*. When you create a routing instance, you also need to specify what type of routing instance it is. For example, if you wanted to create a Layer 2 routing instance, the type would be *virtual-switch*, whereas a Layer 3 routing instance would be a *virtual-router*.

The final piece of virtualization is the separation of *bridge domains* and *learning domains*. A learning domain is simply a database of Layer 2 forwarding information.

Typically, a learning domain is attached to a bridge domain in a 1:1 ratio. A traditional switch will have a learning domain for every bridge domain. For example, VLAN ID 100 would have a single bridge domain and learning domain. The Juniper MX is able to have multiple learning domains within a single bridge domain. This creates interesting scenarios such as creating a single bridge domain that supports a range of VLAN IDs or simply every VLAN ID possible. It might be a bit difficult to wrap your head around this at first, but this book will walk you through every step in the process.

Layer 2 Networking

Let's take a step back and review what exactly a Layer 2 network is. This chapter introduces a lot of new topics related to Layer 2 switching in the MX, and it's critical that you have an expert understanding of the underlying protocols.

Specifically, we'll take a look at Ethernet. A Layer 2 network, also known as the *data link layer* in the seven-layer Open Systems Interconnection (OSI) model, is simply a means to transfer data between two adjacent network nodes. The feature we're most interested in is virtual local area networks (VLANs) and how they're processed.

A *bridge domain* is simply a set of interfaces that share the same flooding, filtering, and forwarding characteristics. A bridge domain and *broadcast domain* are synonymous in definition and can be used interchangeably with each other.

Ethernet II

By default, an Ethernet frame isn't aware of which VLAN it's in as there's no key to unique identify this information. As the frame is flooded, filtered, or forwarded, it's done so within the default bridge domain of the interface. Let's take a look at the format of a vanilla Ethernet II frame.

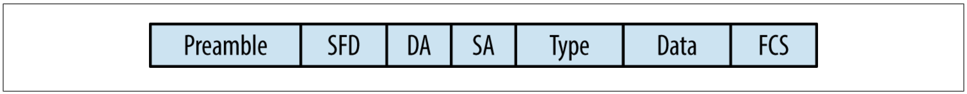


Figure 2-2. Ethernet II frame format

There are seven fields in an Ethernet II frame: preamble, start frame delimiter (SFD), destination address (DA), source address (SA), type, data, and frame check sequence (FCS). Let's take a closer look at each.

Preamble

This eight-octet field is used by hardware to be able to easily identify the start of a new frame. If you take a closer look, it's actually two fields: preamble and SFD. The preamble is seven octets of alternating 0s and 1s. The SFD is a single octet of 1010 1011 to signal the end of the preamble, and the next bit is immediately followed by the destination MAC address.

Destination Address

The destination MAC address is six octets in length and specifies where the Ethernet frame is to be forwarded.

Source Address

The source MAC address is six octets in length and specifies the MAC address from which the Ethernet frame was originally sent.

EtherType

The EtherType is a two-octet field that describes the encapsulation used within the payload of the frame and generally begins at 0x0800. Some of the most common EtherTypes are listed in Table 2-1.

Table 2-1. Common EtherTypes

EtherType	Protocol
0x0800	Internet Protocol Version 4 (IPv4)
0x86DD	Internet Protocol Version 6 (IPv6)
0x0806	Address Resolution Protocol (ARP)
0x8847	MPLS unicast
0x8848	MPLS multicast
0x8870	Jumbo Frames
0x8100	IEEE 802.1Q (VLAN-tagging)
0x88A8	IEEE 802.1QinQ

Payload

This field is the only variable length field in an Ethernet frame. Valid ranges are 46 to 1500 octets, unless Jumbo Frames are being used. The actual data being transmitted is placed into this field.

Frame Check Sequence

This four-octet field is simply a checksum using the cyclic redundancy check (CRC) algorithm. The algorithm is performed against the entire frame by the transmitting node and appended to the Ethernet frame. The receiving node runs the same algorithm and compares it to the FCS field. If the checksum calculated by the receiving node is different than the FCS field on the Ethernet frame, this indicates an error occurred in the transmission of the frame and it can be discarded.

IEEE 802.1Q

Now let's take a look at how it's possible for an Ethernet II frame to suggest which VLAN it would like to be in because, as you will see later in this chapter, it's possible to configure the MX to normalize and change VLAN IDs regardless of the IEEE 802.1Q header. The IEEE 802.1Q standard defines how VLANs are supported within an Ethernet II frame. It's actually an elegant solution because there's no encapsulation performed and only a small four-octet shim is inserted between the Source Address and Type fields in the original Ethernet II frame, as shown in Figure 2-3.

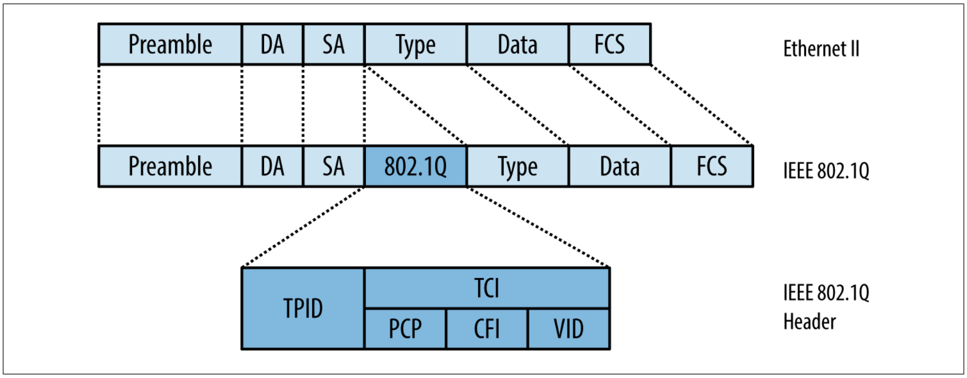


Figure 2-3. Ethernet II and IEEE 802.1Q frame format

This new four-octet shim is divided into two major parts: tag protocol identifier (TPID) and tag control identifier (TCI). The elegant part about the TPID is that it actually functions as a new EtherType field with a value of 0x8100 to indicate that the Ethernet II frame supports IEEE 802.1Q. Notice that both the EtherType in the original Ethernet II frame and the new TPID field in the new IEEE 802.1Q frame begin at the exact same bit position. The TCI is subdivided into three more parts:

Priority Code Point (PCP)
: These three bits describe the frame priority level. This is further defined by IEEE 802.1p.

Canonical Format Indicator (CFI)
: This is a one-bit field that specifies which direction to read the MAC addresses. A value of 1 indicates a noncanonical format, whereas a value of 0 indicates a canonical format. Ethernet and IEEE 802.3 always use a canonical format and the least significant bits first, whereas Token Ring is the opposite and sends the most significant bit first. This is really just an outdated relic and the CFI will always have a value of 0 on any modern Ethernet network.

VLAN Identifier (VID)
: The VID is a 12-bit value that indicates which VLAN the Ethernet frame belongs to. There are 4,094 possible values, as 0x000 and 0xFFF are reserved.

IEEE 802.1QinQ

The next logical progression from IEEE 802.1Q is IEEE 802.1QinQ. This standard takes the same concept of inserting a four-octet shim and expands on it. The challenge is, how do you allow customers to operate their own VLAN IDs inside of the Service Provider's network?

The solution is just as elegant as before. IEEE 802.1QinQ inserts an additional four-octet shim before the IEEE 802.1Q header and after the source MAC address, as shown in Figure 2-4.

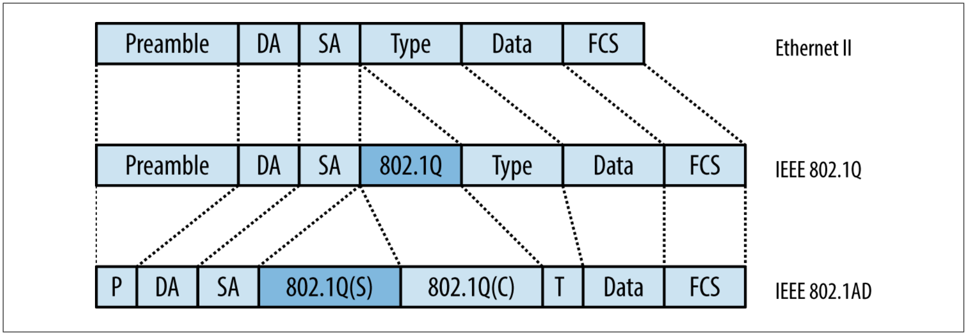


Figure 2-4. IEEE 802.1QinQ frame format

The IEEE 802.1QinQ frame now has two four-octet shims. The first four-octet shim is known as the Service Tag (S-TAG) or outer tag. The second four-octet shim is called the Customer Tag (C-TAG) or inner tag.

The S-TAG has an EtherType of 0x88A8 to signify the presence of an inner tag and indicate that the frame is IEEE 802.1QinQ. The S-TAG is used to provide separation between different customers or Ethernet services.

The C-TAG has an EtherType of 0x8100 to indicate that the frame supports IEEE 802.1Q. The C-TAG represents the original customer VLAN ID.

For example, let's assume there are two customers: Orange and Blue. Let's say that each of the customers internally use the following VLAN IDs:

Orange
  The Orange customer uses VLAN ID 555.

Blue
  The Blue customer uses VLAN ID 1234.

Now let's change the point of view back to the Service Provider, who needs to assign a unique VLAN ID for each customer; say, customer Orange VLAN ID 100 and customer Blue VLAN ID 200.

What you end up with are the following frame formats:

Customer Orange
  S-TAG = 100, C-TAG = 555

Customer Blue
  S-TAG = 200, C-TAG = 1234

This allows Service Providers to provide basic Layer 2 Ethernet services to customers while maintaining the original VLAN IDs set by the customer. However, there a few drawbacks:

Maximum of 4,094 customers or S-TAGs

Because each customer would be mapped to an S-TAG, the maximum number of customers supported would be the 4,094. You can calculate this scaling issue with the following formula: customers = (2^12) – 2. Recall that the VID is 12 bits in width and the values 0x000 and 0xFFF are reserved.

MAC learning

Because the customer destination and source MAC addresses are still present in the IEEE 802.1QinQ frame, all of the equipment in the Service Provider network must learn every host on every customer network. Obviously, this doesn't scale very well, as it could quickly reach the maximum number of MAC addresses supported on the system.

To be fair, a new standard IEEE 802.1AH (MAC-in-MAC) was created to help alleviate the drawbacks listed previously. However, many Service Providers have opted to move straight to MPLS and provide VPLS instead.

Junos Interfaces

Before discussing bridging, a closer look at how Junos handles interfaces is required. Bridging on the MX is fundamentally different than on the EX due to the types of challenges being solved. As you move into the finer points of bridging and virtualization within the MX, it's critical that you have a clear understanding of how interfaces are created and applied within bridge domains, routing instances, and other features.

Let's take a look at a single, generic interface that supports multiple units, families, and addresses in Figure 2-5.

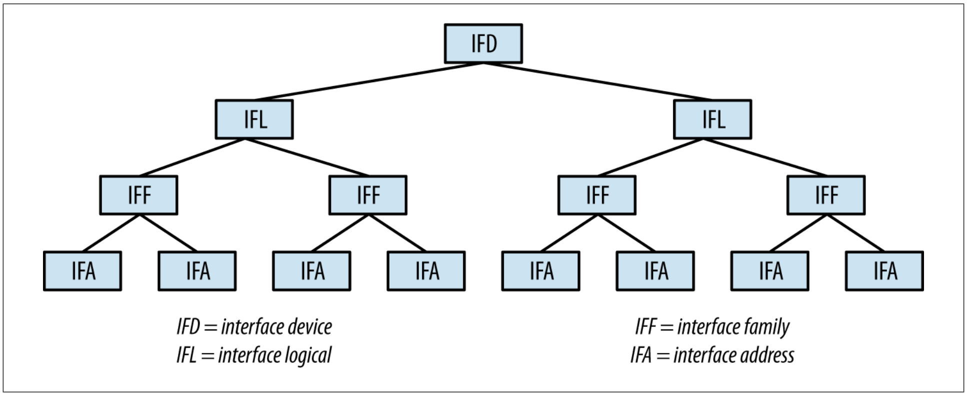


Figure 2-5. Junos interface hierarchy

Interface Device (IFD)
: This represents the physical interface such as xe-0/0/0. This is the root of the hierarchy and all other components are defined and branched off at this point. Features such as maximum transmission unit (MTU), link speed, and IEEE 802.3ad are configured at this level.

Interface Logical (IFL)
: The IFL simply defines a unit number under the IFD such as xe-0/0/0.0 or xe-0/0/0.1. Regardless of the configuration, at least a single unit is required. A common example of multiple IFLs are VLAN ID when using IEEE 802.1Q.

Interface Family (IFF)
: Each IFL needs an address family associated with it, as Junos supports multiple protocols. Common examples include `inet` for IPv4, `inet6` for IPv6, and `iso` when configuring the IS-IS routing protocol.

Interface Address (IFA)
: Finally, each IFF needs some sort of address depending on the type of IFF configured. For example, if the IFF was configured as `inet`, an address might look like 192.168.1.1/24, whereas if the IFF was configured as `inet6`, an address might look like 2001:DB8::1/128.

Let's piece the puzzle together and see what it looks like by configuring the interface xe-0/0/0 with an IPv6 address 2001:DB8::1/128, shown in Figure 2-6.

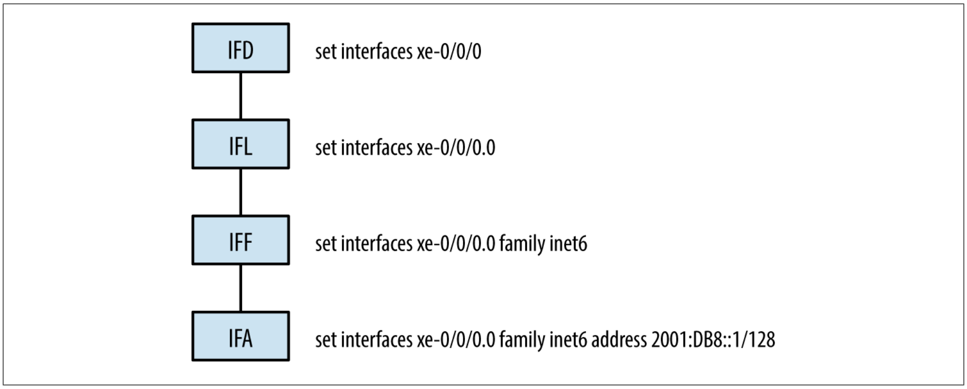

IFD
set interfaces xe-0/0/0
IFL
set interfaces xe-0/0/0.0
IFF
set interfaces xe-0/0/0.0 family inet6
IFA
set interfaces xe-0/0/0.0 family inet6 address 2001:DB8::1/128

Figure 2-6. IFD, IFL, IFD, and IFA example set commands

Given the logical hierarchical structure of the interfaces within Junos, it's easy to see how each section is nicely laid out. This is a perfect example of taking a complex problem and breaking it down into simple building blocks.

Although the Junos interface structure is a good example, the same principles apply throughout the entire design of Junos. It's very natural and easy to understand because it builds upon the good tenets of computer science: divide and conquer with a hierarchical design. For example, enabling IEEE 802.1Q on interface xe-0/0/0 and supporting two VLAN IDs with both IPv4 and IPv6 would look something like Figure 2-7.

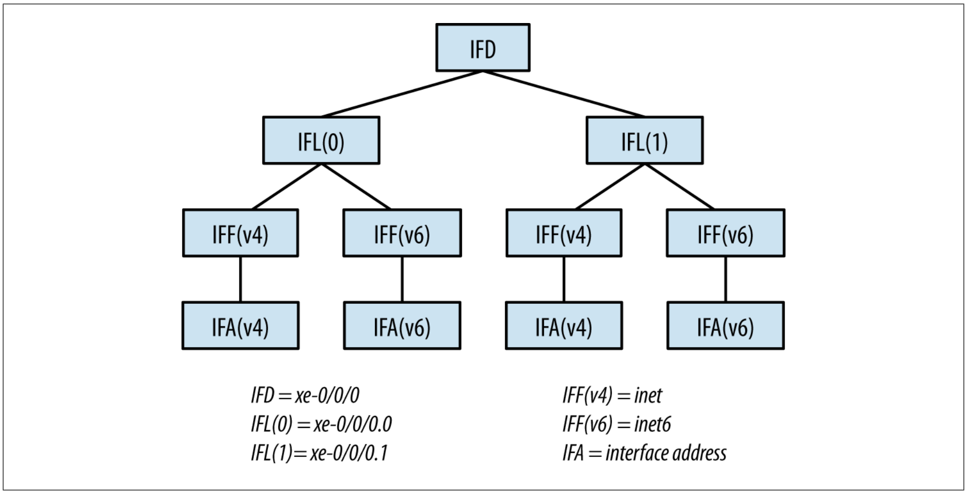


Figure 2-7. IFD, IFL, IFF, and IFA hierarchy with IPv4 and IPv6 families

Interface Bridge Configuration

The MX supports two methods of configuring interfaces for bridging, and each method has its own benefits and drawbacks. One method is geared for more control but requires additional configuration, and the other method is geared toward ease of use but offers less functionality. Both methods are covered in detail.

Service Provider Style
: As mentioned previously in this chapter, Service Providers have very unique challenges in terms of providing both scale and Ethernet-based services to their customers. Such a solution requires extreme customization, flexibility, and scale. The drawback is that when crafting advanced bridging designs, the configuration becomes large. The obvious benefit is that all bridging features are available and can be arranged in any shape and size to provide the perfect Ethernet-based services for customers.

Enterprise Style
: Typical Enterprise users only require traditional switching requirements. This means a single Layer 2 network with multiple bridge domains. Because the requirements are so simple and straightforward, the Juniper MX offers a simplified and condensed method to configure Layer 2 interfaces.

Basic Comparison of Service Provider versus Enterprise Style

Let's start with a very basic example and create two bridge domains and associate two interfaces with each bridge domain, as shown in Figure 2-8, which would require the interfaces to use VLAN tagging.

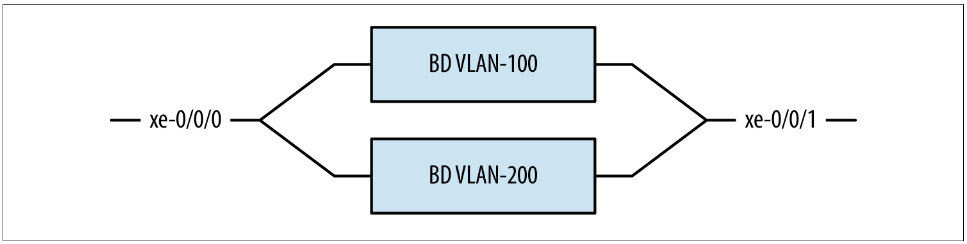


Figure 2-8. Two interfaces and two bridge domains

Service Provider Style

This style requires explicit configuration of each feature on the interface. Once the interfaces are configured, each bridge domain needs to explicitly reference the interface as well. Let's review the interface requirements:

vlan-tagging
: This option modifies the IFD to operate in IEEE 802.1Q mode, also known as a trunk interface.

extended-vlan-bridge
: This encapsulation is applied to the IFD to enable bridging on all IFLs.

unit
: Each VLAN ID that needs to be bridged is required to be broken out into its own IFL. It's common practice to name the unit number the same as the VLAN ID it's associated with.

Armed with this new information, let's take a look at how to configure two interfaces for bridging across two bridge domains:

```
interfaces {
    xe-0/0/0 {
        vlan-tagging;
        encapsulation extended-vlan-bridge;
        unit 100 {
            vlan-id 100;
        }
        unit 200 {
            vlan-id 200;
        }
    }
    xe-0/0/1 {
        vlan-tagging;
        encapsulation extended-vlan-bridge;
        unit 100 {
            vlan-id 100;
        }
        unit 200 {
            vlan-id 200;
        }
    }
}
bridge-domains {
    VLAN-100 {
        vlan-id 100;
        interface xe-0/0/0.100;
        interface xe-0/0/1.100;
    }
    VLAN-200 {
        vlan-id 200;
        interface xe-0/0/0.200;
        interface xe-0/0/1.200;
    }
}
```

As you can see, each IFD has `vlan-tagging` enabled as well as the proper encapsulation, as shown in Figure 2-9. Each VLAN ID is broken out into its own IFL.

The bridge domain configuration is very easy. Each bridge domain is given a name, VLAN ID, and a list of interfaces.

Enterprise Style

This style is very straightforward and doesn't require explicit configuration of every feature, which reduces the amount of configuration required, but as you will see later in this chapter, also limits the number of features this style is capable of.

Let's review the interface requirements:

family
: Each IFL requires `family bridge` to be able to bridge.

interface-mode
: Each IFL requires the `interface-mode` to indicate whether the IFD should operate as an access port (untagged) or trunk (IEEE 802.1Q).

vlan-id
: Each IFL requires a `vlan-id` to specify which bridge it should be part of.

Sounds easy enough. Let's see what this looks like compared to the previous Service Provider style:

```
interfaces {
    xe-0/0/0 {
        unit 0 {
            family bridge {
                interface-mode trunk;
                vlan-id-list [ 100 200 ];
            }
        }
    }
    xe-0/0/1 {
        unit 0 {
            family bridge {
                interface-mode trunk;
                vlan-id-list [ 100 200 ];
            }
        }
    }
}
bridge-domains {
    VLAN-100 {
        vlan-id 100;
    }
    VLAN-200 {
        vlan-id 200;
    }
}
```

That's a pretty big difference. There are no more options for `vlan-tagging`, setting the encapsulation type, or creating an IFL for each VLAN ID. Simply set the `interface-type` to either `access` or `trunk`, set a `vlan-id` or `vlan-id-list`, and you're good to go.

Notice how the illustration in Figure 2-10 has changed to reflect the use of a single IFL per IFD:

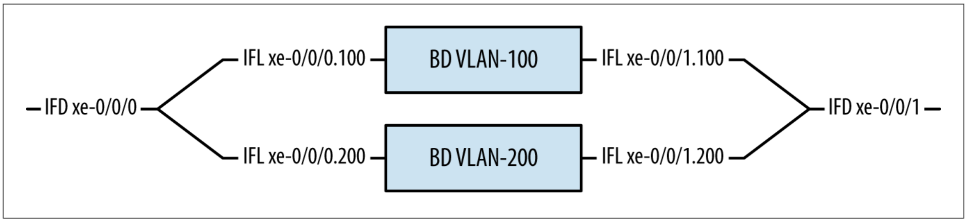


Figure 2-9. Service Provider style

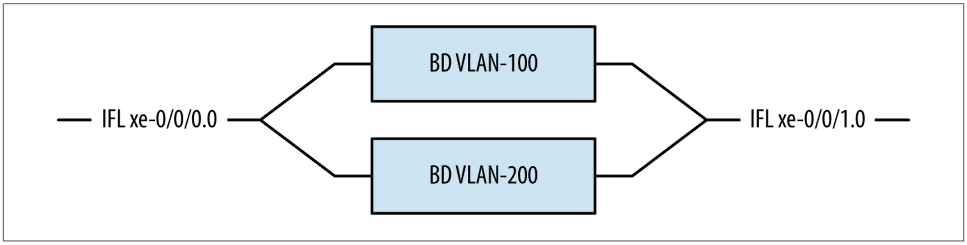


Figure 2-10. Enterprise Style

The best part is that you no longer have to explicitly place interfaces in each bridge domain. When you commit the configuration, Junos will automatically parse the interface structure, identify any interfaces using the Enterprise-style configuration, and place each interface into the appropriate bridge domain based off the `vlan-id` when using IEEE 802.1Q and based-off the inner VLAN IDs when using IEEE 802.1QinQ.

Service Provider Interface Bridge Configuration

It's very common for Service Providers to have multiple customers connected to the same physical port on Provider Equipment (PE). Each customer has unique requirements that require additional features on the PE. For example, a customer may have the following requirements:

- IEEE 802.1Q or 802.1QinQ VLAN mapping
- Class of Service (CoS) based on Layer 4 information, VLAN ID, or IEEE 802.1p
- Acceptable VLAN IDs used by the customer
- Input and output firewall filtering

The Service Provider-style interface configuration is a requirement when dealing with IEEE 802.1Q and 802.1QinQ VLAN mapping or forcing IFLs into a particular bridge domain without relying on Junos to make bridge domain determinations.

Tagging

There are several different types of VLAN tagging available with the Service Provider-style interface configuration. When implementing VLAN tagging, you have the option to support single tag, dual tag, or a combination of both on the same interface. It's required to explicitly define which VLAN tagging is needed on the interface, as there's no automation.

VLAN Tagging

The most basic type of VLAN tagging is the vanilla IEEE 802.1Q, enabled by applying the option `vlan-tagging` to the IFD. Let's take a look:

```
interfaces {
    ae5 {
        vlan-tagging;
        encapsulation extended-vlan-bridge;
        unit 100 {
            vlan-id 100;
        }
        unit 200 {
            vlan-id 200;
        }
        unit 300 {
            vlan-id 300;
        }
    }
}
```

In this example, interface `ae5` will support IEEE 802.1Q for the VLAN IDs 100, 200, and 300. It's required that each VLAN ID has its own IFL unless you're using a special type of bridge domain that doesn't have a VLAN ID. This is more of a corner case that will be reviewed in depth later in this chapter.

> Although `vlan-tagging` enables IEEE 802.1Q, it's important to note that `vlan-tagging` is really just parsing the outermost tag in the frame. For example, if interface `ae5.100` received a frame with a S-TAG of 100 and C-TAG of 200, it would be valid and switched based off the S-TAG, as it matches the `vlan-id` of interface `ae5.100`.

vlan-id-range. There will be times where it makes sense to accept a range of VLAN IDs, but the configuration may be burdensome. The `vlan-id-range` option allows you to specify a range of VLAN IDs and associate it to a single IFL:

```
interfaces {
    ae5 {
        stacked-vlan-tagging;
        encapsulation extended-vlan-bridge;
        unit 0 {
           vlan-id-range 100-500;
        }
    }
}
```

The `vlan-id-range` can only be used with IFLs associated with `bridge-domain vlan-id all`. Any other type of bridge domain doesn't support `vlan-id-range`.

If the goal is to reduce the size of the configuration, and you do not need any advanced VLAN mapping, you might want to consider using the Enterprise-style interface configuration (covered in detail later in the chapter).

Stacked VLAN Tagging

The next logical progression is to support dual tags or IEEE 802.1QinQ. You'll need to use the `stacked-vlan-tagging` option on the IFD to enable this feature:

```
interfaces {

    ae5 {
        stacked-vlan-tagging;
        encapsulation extended-vlan-bridge;

        unit 1000 {
            vlan-tags outer 100 inner 1000;
        }

        unit 2000 {
            vlan-tags outer 100 inner 2000;
        }

        unit 3000 {
            vlan-tags outer 100 inner 3000;
        }
    }
}
```

A couple of things have changed from the previous example with the vanilla IEEE 802.1Q tagging. Notice that the keyword `stacked-vlan-tagging` is applied to the IFD; the other big change is that each IFL is no longer using the `vlan-id` option, but instead now uses the `vlan-tags` option. This allows the configuration of both an `outer` and `inner` tag. The outer tag is often referred to as the S-TAG, and the inner tag is often referred to as the C-TAG.

There is a subtle but very important note with regard to `stacked-vlan-tagging`: *this option isn't required to bridge IEEE 802.1QinQ*. As mentioned previously, the `vlan-tagging` option has no problem bridging IEEE 802.1QinQ frames, but will only use the S-TAG when making bridging decisions. The key difference is that `stacked-vlan-tagging` is required if you want to make bridging decisions based off the inner or C-TAG.

To illustrate the point better, let's expand the bridging requirements. For each C-TAG, let's apply different filtering and policing:

```
interfaces {
    ae5 {
        stacked-vlan-tagging;
        encapsulation extended-vlan-bridge;
        unit 1000 {
            vlan-tags outer 100 inner 1000;
            layer2-policer {
                input-policer 50M;
            }
        }
        unit 2000 {
            vlan-tags outer 100 inner 2000;
             family bridge {
                 filter {
                     input mark-ef;
            }
        }
        unit 3000 {
            vlan-tags outer 100 inner 3000;
             family bridge {
                 filter {
                     input mark-be;
            }
        }
    }
}
```

Now, this is more interesting. Although each frame will have identical S-TAGs (`vlan-id 100`), each C-TAG will be subject to different filtering and policing.

- All traffic with a S-TAG of 100 and C-TAG of 1000 will be subject to a policer.
- All traffic with a S-TAG of 100 and C-TAG of 2000 will be subject to a filter that puts all traffic into the EF forwarding class.
- And finally, all traffic with a S-TAG of 100 and C-TAG of 3000 will be subject to a filter that puts all traffic into the BE forwarding class.

Flexible VLAN Tagging

The final VLAN tagging option combines all of the previous methods together. This option is known as `flexible-vlan-tagging` and allows for both single tag (`vlan-tagging`) and dual tag (`stacked-vlan-tagging`) to be defined per IFL. Let's see it in action:

```
interfaces {
    ae5 {
        flexible-vlan-tagging;
        encapsulation extended-vlan-bridge;
        unit 100 {
            vlan-id 100;
        }
        unit 200 {
            vlan-tags outer 200 inner 2000;
        }
        unit 300 {
            vlan-id 300
        }
    }
}
```

This is pretty straightforward. IFL ae5.100 is bridging based off a single tag, IFL ae5.200 is bridging based off dual tags, and IFL ae5.300 is bridging off a single tag.

It's recommended to use `flexible-vlan-tagging` whenever you need to tag frames. The benefit is that it works with either method of tagging and as you modify and scale your network in the future, you won't run into annoying commit messages about `vlan-tags` not being supported with `vlan-tagging`.

The other great benefit of `flexible-vlan-tagging` is the ability to configure IFLs that have dual tags, but can accept a single tag or untagged frame:

```
interfaces {
    ae5 {
        flexible-vlan-tagging;
        encapsulation extended-vlan-bridge;
        native-vlan-id 100;
        unit 200 {
            vlan-tags outer 100 inner 200;
        }
        unit 300 {
            native-inner-vlan-id 300;
            vlan-tags outer 100 inner 300;
        }
    }
}
```

You can see in this example that `ae5` has a `native-vlan-id` of 100, meaning that any IFL can accept frames with a single tag and the IFL will assume that such frames have a S-TAG of 100. For example, if a frame arrived with a single tag of 200, IFL `ae5.200` would accept it. Taking the example even further, if an untagged frame arrived on `ae5`, IFL `ae5.300` would accept it because it would assume that the S-TAG would be 100 and the C-TAG would be 300. Notice the IFL `ae5.300` has the option `native-inner-vlan-id 300`.

Encapsulation

To be able to support bridging, the Juniper MX requires that the proper encapsulation be applied to the interface. At a high level, there's support for Ethernet bridging, VLAN Layer 2 bridging, cross-connects, and VPLS.

> This book focuses only on Ethernet and VLAN bridging. Cross-connect and VPLS encapsulations types are out of the scope of this book.

Ethernet Bridge

In order to create untagged interfaces, also referred to as *access ports*, the encapsulation type of `ethernet-bridge` needs to be used. Access ports receive and transmit frames without any IEEE 802.1Q and 802.1QinQ shims. The most common use case for untagged interfaces is when you need to directly connect a host or end-device to the MX. Let's take a closer look:

```
interfaces {
    xe-0/0/0 {
        encapsulation ethernet-bridge;
        unit 0;
    }
}
```

Untagged interfaces are deceivingly simple. No need to define a bunch of IFLs or VLAN IDs here. Set the encapsulation to `ethernet-bridge`, and you're good to go. It's almost too easy. But wait a second—if you don't need to define a VLAN ID, how do you put an untagged interface into the right bridge domain? The answer lies within the `bridge-domain` configuration:

```
bridge-domains {
    vlan-100 {
        domain-type bridge;
        vlan-id 100;
        interface xe-0/0/0.0;
    }
}
```

When you configure a bridge domain, simply associate the untagged interface into the bridge domain. When using the Service Provider-style configuration, it's always required that you manually place every single IFL you want to be bridged into a bridge domain. This is covered in detail later in the chapter.

When using an IFD encapsulation of `ethernet-bridge`, it requires a single IFL of 0 to be created. No other IFL number is valid.

Extended VLAN Bridge

When you're configuring an interface for VLAN tagging, it doesn't matter what type of tagging you use because there's a single encapsulation to handle them all: `extended-vlan-bridge`. This encapsulation type has already been used many times in previous examples. Let's take a look at one of those previous example and review the encapsulation type:

```
interfaces {
    ae5 {
        flexible-vlan-tagging;
        encapsulation extended-vlan-bridge;
        unit 100 {
            vlan-id 100;
        }
        unit 200 {
            vlan-tags outer 200 inner 2000;
        }
        unit 300 {
            vlan-id 300;
        }
    }
}
```

When using the extended-vlan-bridge encapsulation type, it can only be applied to the IFD. The added benefit when using extended-vlan-bridge is that it automatically applies the encapsulation vlan-bridge to all IFLs, so there's no need to apply an encapsulation for every single IFL.

Flexible Ethernet Services

Providing Ethernet-based services to many customers on the same physical interface creates some interesting challenges. Not only do the services need to be isolated, but the type of service can vary from vanilla bridging, VPLS bridging, or a Layer 3 handoff. The MX provides an IFD encapsulation called flexible-ethernet-services to provide per IFL encapsulation, allowing each IFL to independently provide Ethernet-based services. Consider this:

```
    interfaces {
        ae5 {
            flexible-vlan-tagging;
            encapsulation flexible-ethernet-services;
            unit 100 {
                encapsulation vlan-bridge;
                vlan-id 100;
            }
            unit 200 {
                encapsulation vlan-vpls;
                vlan-id 200;
            }
            unit 1000 {
                encapsulation vlan-bridge;
                vlan-tags outer 300 inner 1000;
            }
            unit 3000 {
                vlan-id 3000;
                family inet6 {
                    address 2001:db8:1:3::0/127;
                }
            }
        }
    }
```

This example is interesting because each IFL is providing a different type of Ethernet-based service. Three of the IFLs are associated with bridge domains, whereas the last

IFL is simply a routed interface associated with the `inet6.0` route table. Flexible Ethernet services is illustrated in Figure 2-11.

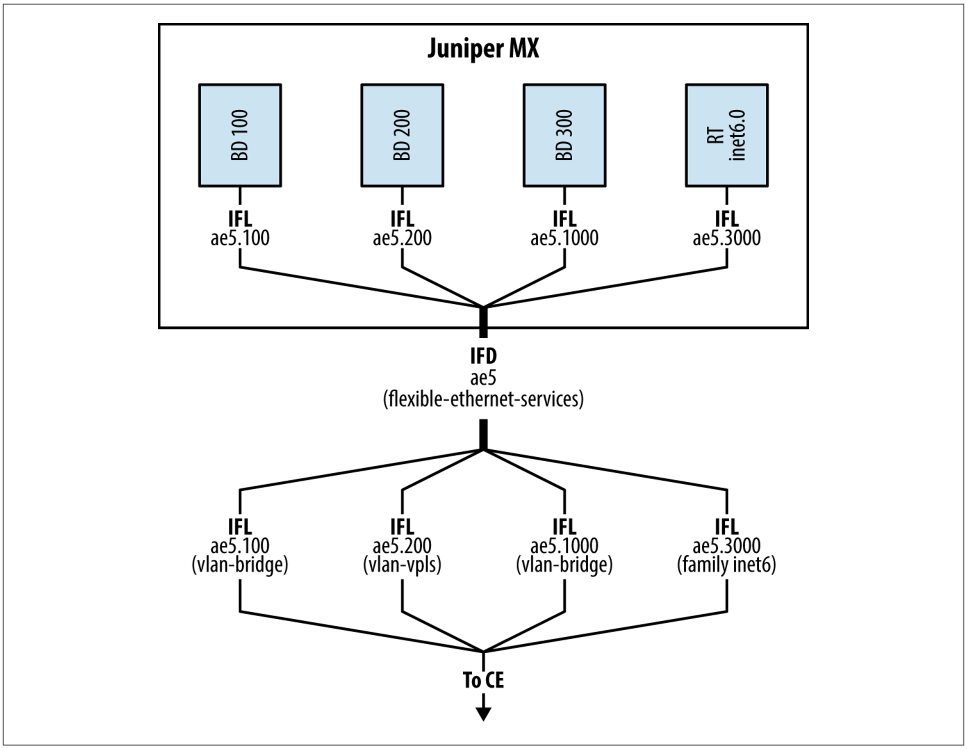


Figure 2-11. Illustration of Flexible Ethernet Services.

Let's walk through each IFL in Figure 2-11 and review what is happening there.

IFL ae5.100
: This is a vanilla IEEE 802.1Q configuration accepting frames with the `vlan-id` of 100. Notice the IFL has an encapsulation of `vlan-bridge` to accept frames with a TPID of 0x8100. Without this IFL encapsulation, it wouldn't be able to bridge.

IFL ae5.200
: This is another vanilla IEEE 802.1Q configuration accepting frames with the `vlan-id` of 200. The big difference here is that this IFL is part of a VPLS routing instance and uses an IFL encapsulation of `vlan-vpls`.

IFL ae5.1000
: This is a bit more advanced, but nothing you haven't seen before. Allowing dual tags is actually a function of the IFD option `flexible-vlan-tagging`, but it's included here to make the example more complete. This IFL also requires the encapsulation type of `vlan-bridge` so that it's able to bridge.

IFL ae5.3000

This is the odd man out because no bridging is going on here at all, and thus no IFL encapsulation is required. IEEE 802.1Q is being used on this IFL with a `vlan-id` of 3000 and providing IPv6 services via `family inet6`.

An interesting side effect of `flexible-ethernet-services` is that you can combine this with the `native-vlan-id` option to create an untagged interface as well. The only caveat in creating a pure untagged port is that it can only contain a single IFL and the `vlan-id` must match the `native-vlan-id` of the IFD:

```
interfaces {
    ae5 {
        flexible-vlan-tagging;
        native-vlan-id 100;
        encapsulation flexible-ethernet-services;
        unit 0 {
            encapsulation vlan-bridge;
            vlan-id 100;
        }
    }
}
```

This configuration allows the interface `ae5.0` to receive and transmit untagged frames. As always, there is more than one way to solve a challenge in Junos, and no one way is considered wrong.

Service Provider Bridge Domain Configuration

The final piece of configuring Service Provider-style bridging is associating interfaces with bridge domains. It isn't enough to simply refer to the `vlan-id` in the IFL configuration. Each IFL needs to be included as part of a bridge domain in order to bridge properly. Simply use the `interface` knob under the appropriate `bridge domain` to enable bridging.

Let's take a look at the previous Flexible Ethernet Services interface configuration and place it into some bridge domains:

```
interfaces {
    ae5 {
        flexible-vlan-tagging;
        encapsulation flexible-ethernet-services;
        unit 100 {
            encapsulation vlan-bridge;
            vlan-id 100;
        }
        unit 200 {
            encapsulation vlan-vpls;
            vlan-id 200;
        }
        unit 1000 {
            encapsulation vlan-bridge;
            vlan-tags outer 300 inner 1000;
        }
        unit 3000 {
            vlan-id 3000;
            family inet6 {
                address 2001:db8:1:3::0/127;
            }
        }
    }
}
```

There are three IFLs that need to be bridged: ae5.100, ae5.200, and ae5.1000. Let's create a bridge domain for each and install the IFLs into each respective bridge domain:

```
bridge-domains {
    bd-100 {
        vlan-id 100;
        interface ae5.100;
    }
    bd-200 {
        vlan-id 200;
        interface ae5.200;
    }
    bd-300 {
        vlan-id 300;
        interface ae5.1000;
    }
}
```

To make it more interesting let's create an untagged interface and place it into bridge domain `bd-100`:

```
interfaces {
    xe-0/0/0 {
        encapsulation ethernet-bridge;
        unit 0;
    }
}
```

Now that there's an access port, you still have to install it into bridge domain `bd-100`:

```
bridge-domains {
    bd-100 {
        vlan-id 100;
        interface ae5.100;
        interface xe-0/0/0.0;
    }
    bd-200 {
        vlan-id 200;
        interface ae5.200;
    }
    bd-300 {
        vlan-id 300;
        interface ae5.1000;
    }
}
```

Simple enough. Create an interface, configure the appropriate IFLs, set the `vlan-id`, and install it into a bridge domain. Although simple, every good engineer wants to know what's happening behind the covers, so let's inspect the interface for some more clues:

```
dhanks@R2-RE0> show interfaces ae5.100
  Logical interface ae5.100 (Index 326) (SNMP ifIndex 574)
    Flags: SNMP-Traps 0x24004000 VLAN-Tag [ 0x8100.100 ]  Encapsulation: VLAN-Bridge
    Statistics        Packets        pps         Bytes          bps
    Bundle:
        Input :        981977          1      99889800          816
        Output:        974005          0      99347628            0
    Protocol bridge, MTU: 1518
```

There you go, and the most important bit is `VLAN-Tag [ 0x8100.100 ]`. You can see the TPID and VLAN ID associated with this IFL.

What's stopping you from placing interface `ae5.100` into a bridge domain with a mismatched VLAN ID? Currently, `ae5.100` is configured with a `vlan-id` of 100, so let's move it into a bridge domain with a `vlan-id` of 200:

```
bridge-domains {
    bd-100 {
        vlan-id 100;
        interface xe-0/0/0.0;
    }
    bd-200 {
        vlan-id 200;
        interface ae5.100;
        interface ae5.200;
    }
    bd-300 {
        vlan-id 300;
        interface ae5.300;
    }
}
```

This should be interesting. Let's take another look at the interface:

```
dhanks@R2-RE0> show interfaces ae5.100
  Logical interface ae5.100 (Index 324) (SNMP ifIndex 729)
    Flags: SNMP-Traps 0x24004000 VLAN-Tag [ 0x8100.100 ] In(swap .200) Out(swap .100)
  Encapsulation: VLAN-Bridge
    Statistics        Packets        pps         Bytes          bps
    Bundle:
        Input :             4          0           272            0
        Output:            38          0          2992            0
    Protocol bridge, MTU: 1518
```

Very interesting indeed! The `VLAN-Tag` has changed significantly. It's now showing `0x8100.100` with some additional `In` and `Out` operations. This is called *VLAN normalization* or *rewriting*, and it's covered in more detail later in the chapter.

Enterprise Interface Bridge Configuration

Enterprise users generally have very basic requirements compared to Service Providers customers. Instead of providing Ethernet-based services, they are consuming Ethernet-based services. This means that each port is connected to a host or providing a trunk to another switch.

Because of the simplicity of these requirements, it wouldn't be fair to enforce the same configuration. Instead, the Juniper MX provides an Enterprise-style interface configuration that provides basic bridging functionality with a simplified configuration.

Some of the obvious drawbacks of a simplified configuration are the loss of advanced features like VLAN normalization. But the big benefit is that you no longer have to worry about the different VLAN tagging or encapsulation options. When you commit a configuration, Junos will automatically walk the interface tree and look for IFLs using the Enterprise-style interface configuration and determine what VLAN tagging and encapsulation options are needed.

The icing on the cake is that you no longer have to specify which bridge domain each interface belongs to; this happens automatically on commit, but it doesn't actually modify the configuration. Junos will walk the interface tree and inspect all of the Enterprise-style IFLs, look and see which VLAN IDs have been configured, and automatically place the IFLs into the matching bridging domain. Now you can see why advanced VLAN normalization isn't possible.

Interface Mode

The Enterprise-style interface configuration revolves around the `interface-mode` option that places the IFL in either an *access* or *trunk* mode. For readers familiar with the Juniper EX switches, these options will look very similar.

Access

To create an untagged interface, you need to use the `interface-mode` with the `access` option, and you will also need to specify which VLAN the access port will be part of using the `vlan-id` option. Let's see what this looks like:

```
interfaces {
    xe-0/0/0 {
        unit 0 {
            family bridge {
                interface-mode access;
                vlan-id 100;
            }
        }
    }
}
bridge-domains {
    VLAN-100 {
        vlan-id 100;
    }
}
```

Attention to detail is critical when setting the `vlan-id` in access mode. It must be applied on the IFF under `family bridge`. If you try setting the `vlan-id` on the IFL, you will get an error.

- CORRECT

  ```
  set interfaces xe-0/0/0.0 family bridge vlan-id 100
  ```

- INCORRECT

  ```
  set interfaces xe-0/0/0.0 vlan-id 100
  ```

You can see that this is much easier. No more encapsulations, VLAN tagging options, or having to put the IFL into the appropriate bridge domain. The Enterprise style is very straightforward and resembles the EX configuration. Table 2-2 is an Enterprise cheat sheet for the MX versus the EX configuration style.

Table 2-2. MX versus EX Interface Configuration Cheat Sheet.

MX	EX
`family bridge`	`family ethernet-switching`
`interface-mode`	`port-mode`
`vlan-id`	`vlan members`
`vlan-id-list`	`vlan members`

Trunk

The other option when using `interface-mode` is trunk mode, which creates an IEEE 802.1Q IFL. And there's no need to fiddle with VLAN tagging options, as the `interface-mode` will take care of this automatically:

```
interfaces {
    xe-0/0/0 {
        unit 0 {
            family bridge {
                interface-mode trunk;
                vlan-id-list 1-4094;
            }
        }
    }
}
bridge-domains {
    VLAN-100 {
        vlan-id 100;
    }
    VLAN-200 {
        vlan-id 200;
```

```
        }
    }
```

To define which bridge domains the trunk participates in, you need to use the vlan-id-list option and specify the VLAN IDs. Also note there's no need to include the IFL in a specific bridge-domain, as Junos will do this automatically and match up IFLs and bridge domains based off the vlan-id. In this specific example, the IFL xe-0/0/0.0 has a vlan-id-list that includes all possible VLANs, so it will be part of both bridge domains VLAN-100 and VLAN-200. It's acceptable to have a single VLAN ID on the IFL.

IEEE 802.1QinQ

The Enterprise-style interface configuration also supports dual tags. The S-TAG is defined with the IFL option vlan-id and the C-TAG is defined with the IFF option inner-vlan-id-list. Let's review:

```
    interfaces {
        xe-2/1/1 {
            flexible-vlan-tagging;
            unit 0 {
                vlan-id 100;
                family bridge {
                    interface-mode trunk;
                    inner-vlan-id-list [ 1000 2000 ];
                }
            }
        }
    }
```

One important thing to remember is that flexible-vlan-tagging is required when creating IFLs that support IEEE 802.1QinQ with the Enterprise-style interface configuration. In this example, the S-TAG is VLAN ID 100, and the C-TAG supports either VLAN ID 1000 or 2000.

When Junos walks the interface tree and determines which bridge domain to place the IFL with the Enterprise-style interface configuration, all IFLs with dual tags will be placed into a bridge domain based off of their C-TAG(s) or IFL vlan-id. In this example, Junos would place xe-2/1/1.0 into the bridge domain that was configured with VLAN ID 1000 and 2000.

IEEE 802.1Q and 802.1QinQ Combined

Using an IFD encapsulation of flexible-ethernet-services, you can combine IEEE 802.1Q and 802.1QinQ on a single interface.

```
    interfaces {
        xe-1/0/7 {
            description "IEEE 802.1Q and 802.1QinQ";
            flexible-vlan-tagging;
            encapsulation flexible-ethernet-services;
            unit 100 {
                description "IEEE 802.1Q";
                encapsulation vlan-bridge;
                family bridge {
                    interface-mode trunk;
                    vlan-id-list [ 100 ];
                }
            }
            unit 1000 {
                description "IEEE 802.1QinQ";
                encapsulation vlan-bridge;
                vlan-id 200;
                family bridge {
                    interface-mode trunk;
                    inner-vlan-id-list [ 1000 1001 ];
                }
            }
        }
    }
    bridge-domains {
        VLAN100 {
            vlan-id 100;
        }
        VLAN1000 {
            vlan-id 1000;
        }
        VLAN1001 {
            vlan-id 1001;
        }
    }
```

In this example, there are two IFLs. The first unit 100 is configured for IEEE 802.1Q and will be automatically placed into the bridge domain `VLAN100`. The second unit 1000 is configured for IEEE 802.1QinQ and will be automatically placed into bridge domains `VLAN1000` and `VLAN1001`.

```
dhanks@R1> show bridge domain

Routing instance        Bridge domain            VLAN ID     Interfaces
default-switch          VLAN100                  100         xe-1/0/7.100
default-switch          VLAN1000                 1000        xe-1/0/7.1000
default-switch          VLAN1001                 1001        xe-1/0/7.1000
```

It's always a bit tricky dealing with IEEE 802.1QinQ because you have to remember that the IFL will automatically be placed into bridge domains based off the C-TAG(s) and not the S-TAG of the IFL. In this example, you can see that `xe-1/0/7.1000` is in two bridge domains based off its C-TAGs of 1000 and 1001.

VLAN Rewrite

Imagine a scenario where a downstream device is using IEEE 802.1Q and you need to integrate it into your existing network. The problem is that the VLAN ID being used by the downstream device conflicts with an existing VLAN ID being in your network and you're unable to modify this device. The only device you have access to is the

Juniper MX. One of the solutions is to simply change the VLAN to something else. This is referred to as *VLAN rewriting* or *normalization*.

The Enterprise-style interface configuration supports very basic VLAN rewriting on trunk interfaces. The only caveat is that you can only rewrite the outer tag. Let's create a scenario where you need to rewrite VLAN ID 100 to 500 and vice versa:

```
interfaces {
    ae0 {
        unit 0 {
            family bridge {
                interface-mode trunk;
                vlan-id-list 500;
                vlan-rewrite {
                    translate 100 500;
                }
            }
        }
    }
    ae1 {
        unit 0 {
            family bridge {
                interface-mode trunk;
                vlan-id-list 500;
            }
        }
    }
}
```

The keyword in the configuration is `vlan-rewrite`. You're able to specify multiple `translate` actions, but for this example you only need one. What you want to do is accept frames with the VLAN ID of 100, then translate that into VLAN ID 500. The opposite is true for frames that need to be transmitted on `ae0.0`: translate VLAN ID 500 to VLAN ID 100. An example of VLAN rewrite is illustrated in Figure 2-12.

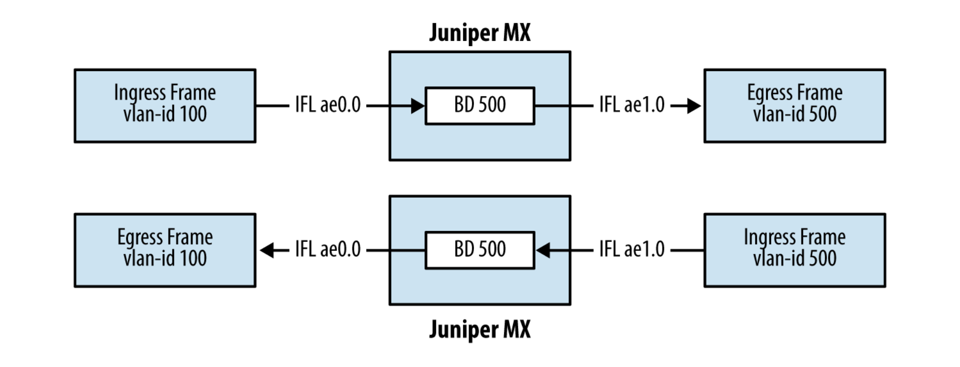


Figure 2-12. Example of Enterprise-style VLAN rewrite on interface ae0.0.

Keep in mind this will only be applied on a per-IFL basis and will not impact other IFDs or IFLs. As mentioned previously, it's possible to have multiple VLAN translations, so let's also translate VLAN ID 101 to VLAN ID 501 on top of our current configuration:

```
interfaces {
    ae0 {
        unit 0 {
            family bridge {
                interface-mode trunk;
                vlan-id-list 500;
                vlan-rewrite {
                    translate 100 500;
                    translate 101 501;
                }
            }
        }
    }
}
```

You can see it's as simple as appending another `translate` operation to `vlan-rewrite`, but sometimes it's easy to get things mixed up, so keep in mind that the VLAN rewrite format is always `vlan-rewrite translate` *`<from> <to>`*.

Service Provider VLAN Mapping

Now that you have a taste for some basic VLAN rewriting, let's expand on the topic. Sometimes changing just the S-TAG of a frame isn't enough. Service Providers provide Ethernet services to many customers, and each customer has their own unique set of requirements. When you need to go beyond doing a simple VLAN rewrite, you have to come back to the Service Provider-style configuration.

Stack Data Structure

Service Provider VLAN mapping allows you to modify the packet in many different ways. Because IEEE 802.1Q and IEEE 802.1QinQ was designed to simply insert a four-octet shim into the frame, you can leverage a computer science data structure called a *stack*. A stack can best be characterized by "last in, first out."

Figure 2-13 illustrates the basic concept of a stack. As items are added to the stack, they are pushed further down. Items can only be removed from the top of the stack. Hence, the last item in is the first out, and the first in is the last out.

There are three basic operations that are used with a stack data structure:

Push
    This operation adds data to the top of the stack.

Pop
    This operation will remove the top of the stack.

Swap
: This operation will swap/exchange the top of the stack with new data.

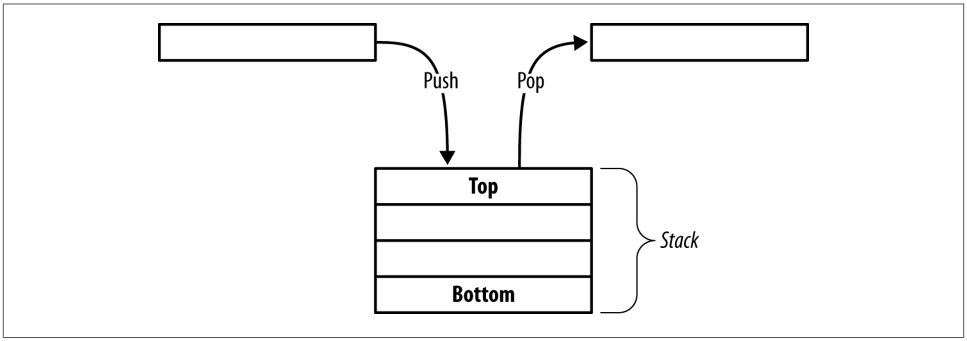


Figure 2-13. Basic stack data structure

When applying the stack data structure to VLAN mapping, the stack becomes the stack of IEEE 802.1Q shims. As new tags are applied to the frame, the last tag to be added will be at the top of the stack. The reverse is true when removing tags from the frame. The last frame to be added will be the first tag to be removed, while the first tag that was added will be the last to be removed.

Stack Operations

The MX supports eight different stack operations, which may seem a bit awkward because a basic stack supports three operations: push, pop, and swap. The answer lies with IEEE 802.1QinQ, as it's very common to manipulate frames with two tags. Five new operations have been added to allow the operator to manipulate two tags at once.

Let's walk through each operation to understand how it manipulates the frame. As you walk through each operation, keep in mind that the top of stack is considered the tag and the bottom of the stack is considered the innermost tag in the frame, as noted in Figure 2-14:

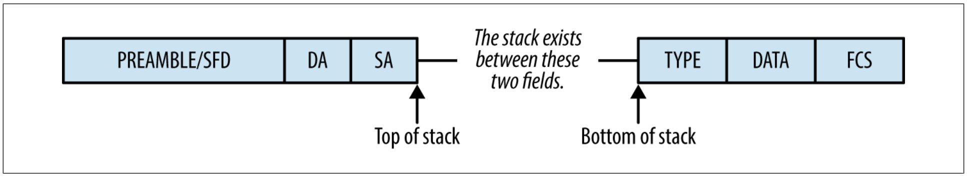


Figure 2-14. Illustrating the IEEE 802.1Q stack in an Ethernet frame

push
: This will simply push a new tag onto the frame. This tag will become the new outer tag, as shown in Figure 2-15.

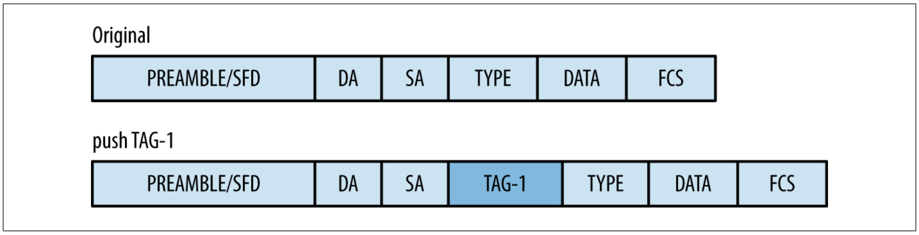


Figure 2-15. Push operation

pop
: This will remove the outermost tag, as shown in Figure 2-16.

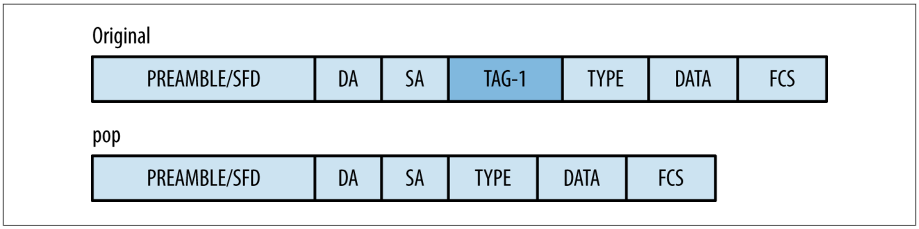


Figure 2-16. Pop operation

swap
: This will swap/exchange the outermost tag with a new user-specified tag, as in Figure 2-17.

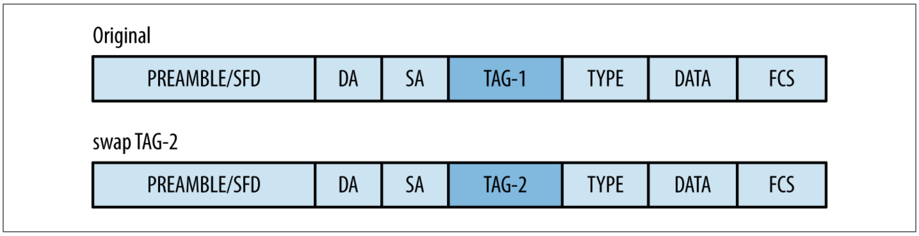


Figure 2-17. Swap operation

push-push
: This operation is very similar to push, except that this will push two user-specified tags to the frame, as shown in Figure 2-18.

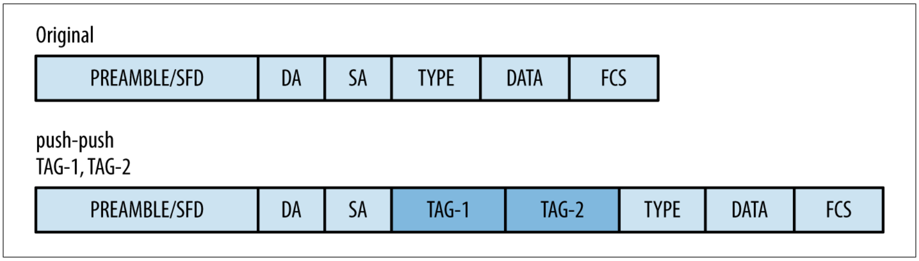


Figure 2-18. Push-push operation

pop-pop
: This will remove the two outermost tags on the frame, as in Figure 2-19.

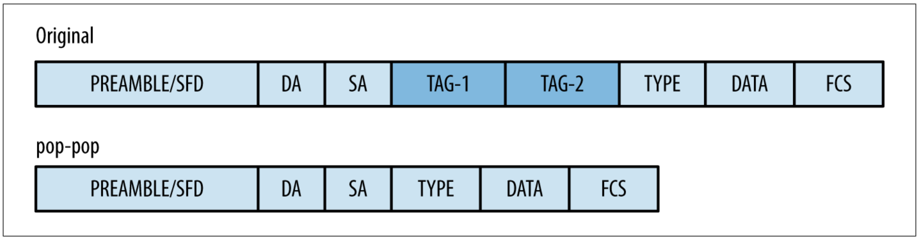


Figure 2-19. Pop-pop operation

swap-swap
: This will swap the two outermost tags with user-specified tags, as in Figure 2-20.

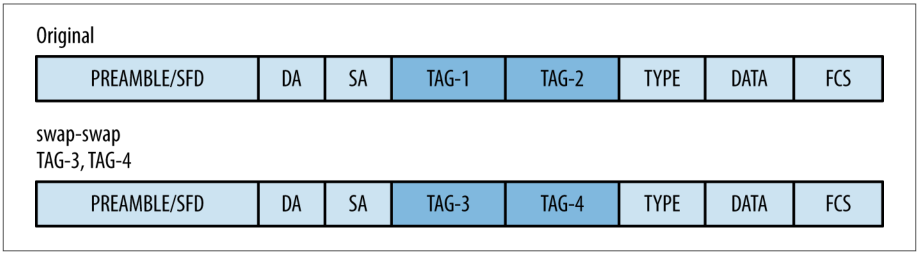


Figure 2-20. Swap-swap operation

swap-push
: This is a tricky one but only at first. It's actually two operations combined as one. The first operation is to swap/exchange the outermost tag with a user-specified tag, and the next operation is to push a new user-specified tag to the frame, as shown in Figure 2-21.

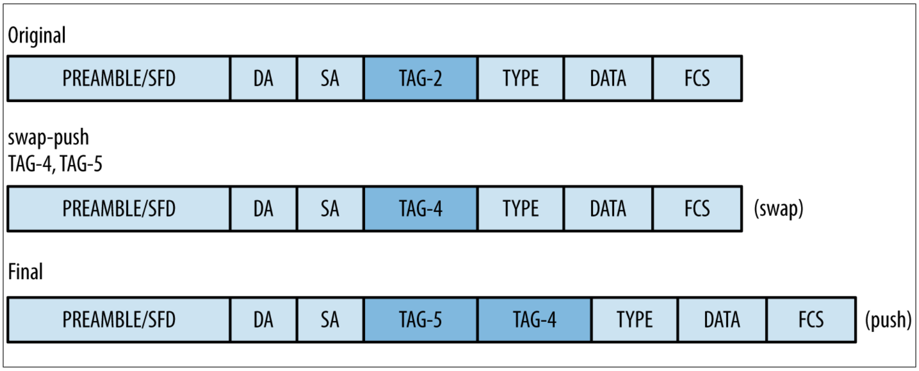


Figure 2-21. Swap-push operation

pop-swap

Here's another tricky one. This operation is two operations combined as one. The first operation removed the outermost tag from the frame, and the second operation is to swap/exchange the outer most tag with a user-specified tag, as shown in Figure 2-22.

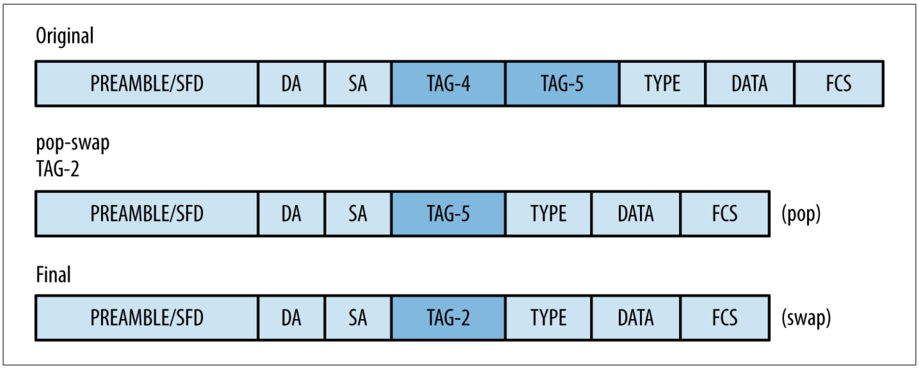


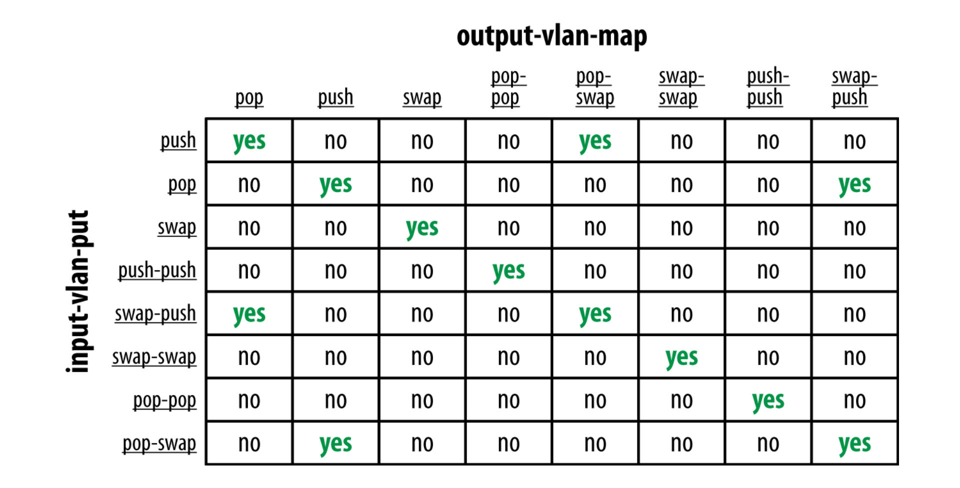
Figure 2-22. Pop-swap operation

This may all seem a bit daunting, due to the new stack operations and operating on two tags at once, but there is a method to the madness. The key to any complex problem is to break it down into simple building blocks, and the simplest form of advanced VLAN mapping is to fully understand how each stack operation modifies a packet and nothing more. The pieces will then be put back together again so you can see the solution come full circle.

Stack Operations Map

Let's understand how these operations can be used in conjunction with each other to perform interesting functions. Each function is designed to be paired with another.

Even more interesting is how the functionality changes depending on the traffic flow. Let's take a look and see how these stack operations can be paired together in different directions, first by reviewing Figure 2-23.



		output-vlan-map							
		pop	push	swap	pop-pop	pop-swap	swap-swap	push-push	swap-push
input-vlan-put	push	yes	no	no	no	yes	no	no	no
	pop	no	yes	no	no	no	no	no	yes
	swap	no	no	yes	no	no	no	no	no
	push-push	no	no	no	yes	no	no	no	no
	swap-push	yes	no	no	no	yes	no	no	no
	swap-swap	no	no	no	no	no	yes	no	no
	pop-pop	no	no	no	no	no	no	yes	no
	pop-swap	no	yes	no	no	no	no	no	yes

Figure 2-23. Stack operations

You can see that each stack operation is designed to be paired up with another stack operation. For example, if you push something onto a stack, the opposite action would be to pop it. Push and pop are considered complementary pairs. There are two types of VLAN maps that deal with the direction of the frame: `input-vlan-map` and `output-vlan-map`.

Think of the two functions `input-vlan-map` and `output-vlan-map` as invertible functions, as shown in Figure 2-24, where:

$$\boxed{f(x) = y \iff g(y) = x}$$

Figure 2-24. Invertible function

This is only true if `output-vlan-map` isn't given any explicit arguments. The default behavior for the function `output-vlan-map` is to use the `vlan-id` or `vlan-tags` of the corresponding IFL.

input-vlan-map

The `input-vlan-map` is a function that's applied to an IFL to perform VLAN mapping to ingress frames. There are five options that can be applied to `input-vlan-map`:

Stack operation
: This is the stack operation to be applied to the ingress frame: `pop`, `swap`, `push`, `push-push`, `swap-push`, `swap-swap`, `pop-swap`, or `pop-pop`.

tag-protocol-id
: This is an optional argument that can be used to set the TPID of the outer tag. For example, `tag-protocol-id` 0x9100 and `vlan-id` 200 would effectively create a 0x9100.200 tag.

vlan-id
: This is the user-specified VLAN ID used for the outer tag.

inner-tag-protocol-id
: This argument is very similar to `tag-protocol-id`. The difference is that this sets the TPID for the inner tag. This is required when using the stack operation `swap-push`.

inner-vlan-id
: This argument is the user-specified inner VLAN ID and is required when using the stack operations `push-push`, `swap-push`, and `swap-swap`.

output-vlan-map

The output-vlan-map is also a function that's applied to an IFL. The difference is that this function is used to apply VLAN mapping to egress frames.

Stack operation
: This is the stack operation to be applied to the egress frame: `pop`, `swap`, `push`, `push-push`, `swap-push`, `swap-swap`, `pop-swap`, or `pop-pop`. This must be complementary to the `input-vlan-map` stack operation. Please refer to Figure 2-23 to see which stack operations can be used with each other.

tag-protocol-id
: This is an optional argument that can be used to set the TPID of the outer tag. If no argument is given, the `output-vlan-map` will use the TPID of the IFL to which it's applied.

vlan-id
: This is the user-specified VLAN ID used for the outer tag. If no argument is given, the `output-vlan-map` will use the `vlan-id` of the IFL to which it's applied.

inner-tag-protocol-id
: This is another optional argument that is very similar to `tag-protocol-id`. The difference is that this sets the TPID for the inner tag. If this argument is omitted, the TPID of the IFL will be used.

inner-vlan-id
: This argument is the user-specified inner VLAN ID and is required when using the stack operations `push-push`, `swap-push`, and `swap-swap`. Again, if no `inner-vlan-id`

is specified, `output-vlan-map` will use the `vlan-tags` inner on the IFL to which it's applied.

There is a lot of confusion regarding the `output-vlan-map` function. It's functionally different from `input-vlan-map` and doesn't require you to specify a `vlan-id` or `inner-vlan-id`. The default is to use the `vlan-id` or `vlan-tags` of the IFL.

The default behavior guarantees that the `output-vlan-map` will translate the egress frame back into its original form. The only reason you should ever specify a `vlan-id` or `inner-vlan-id` in `output-vlan-map` is in a corner case where you require the egress frame to have a different VLAN ID than received.

Tag Count

With many different stack operations available, it becomes difficult to keep track of how many tags were modified and which frames can be used with which stack operations. Table 2-3 should assist you in this case. For example, it doesn't make any sense to use the pop stack operation on an untagged frame.

Table 2-3. Incoming frame: Change in number of tags per stack operation

Operation	Untagged	Single Tag	Two Tags	Change in Number of Tags
pop	no	yes	yes	−1
push	yes[1]	yes	yes	+1
swap	no	yes	yes	0
push-push	yes[1]	yes	yes	+2
swap-push	no	yes	yes	+1
swap-swap	no	no	yes	0
pop-swap	no	no	yes	−1
pop-pop	no	no	yes	−2

All of the stack operations are available when there's an incoming frame with two tags. When an ingress frame with a single tag is received, there's a small subset of stack operations that aren't allowed because they require at least two tags on a frame.

[1] The rewrite operation isn't supported on untagged IFLs. However this will work on tagged interfaces using a `native-vlan-id` and receiving an untagged frame.

The MX doesn't care if the incoming frame has more than two tags. It's completely possible to use all of these stack operations on an Ethernet frame with three tags. The stack operations will continue to work as advertised and modify the first and second tags, but not touch the third tag—unless you force a scenario where you either push or pop a third tag manually.

Bridge Domain Requirements

Service Provider-style VLAN mapping can only be used with a default `bridge-domain`. A default bridge domain doesn't specify any type of `vlan-id` or `vlan-tags`. All of the VLAN ID mappings are required to be performed manually per IFL using `input-vlan-map` and `output-vlan-map`.

Let's take a look at what a default bridge domain looks like with only two interfaces:

```
bridge-domains {
    domain-type bridge;
    interface ae0.100;
    interfaces ae1.100;
}
```

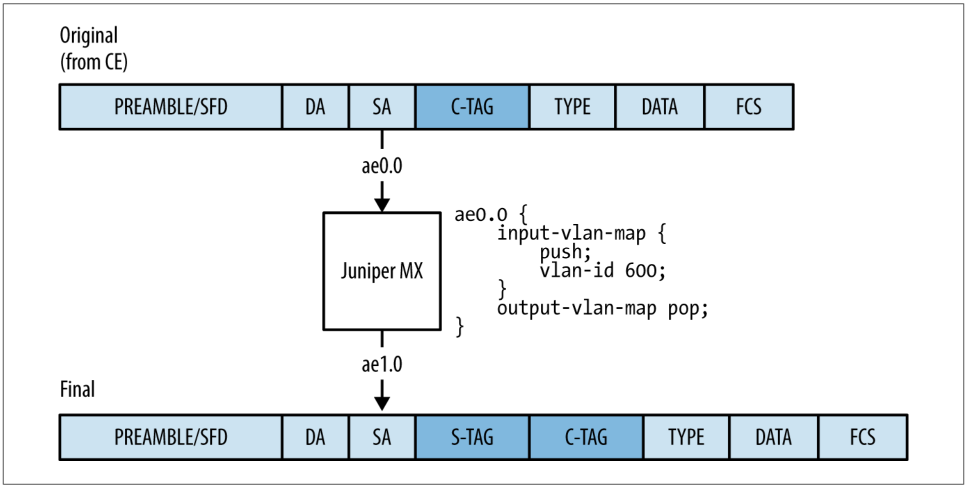
It's so simple, it's actually a bit deceiving and counterintuitive. Network engineers are trained to always include a VLAN ID when creating bridge domains and VLANs—it just seems so natural.

But the secret here is that when you define a `vlan-id` or `vlan-tags` inside of a `bridge-domain`, it's actually just a macro for creating automatic VLAN mappings. Therefore, when you use a `bridge-domain` with a defined `vlan-id`, it's simply a VLAN mapping macro to normalize all VLAN IDs to that specific VLAN ID. For example, if you create a `bridge-domain` called *APPLICATION* with a `vlan-id` of 100, and include an interface of `ae.200` with a `vlan-id` of 200, all frames on interface `ae.200` will be swapped with VLAN ID 100.

This may be a bit confusing at first, but it's covered in depth later in the chapter. For right now, just remember that when using Service Provider-style VLAN mapping, it's required to use a default bridge domain without defining a `vlan-id` or `vlan-tags`.

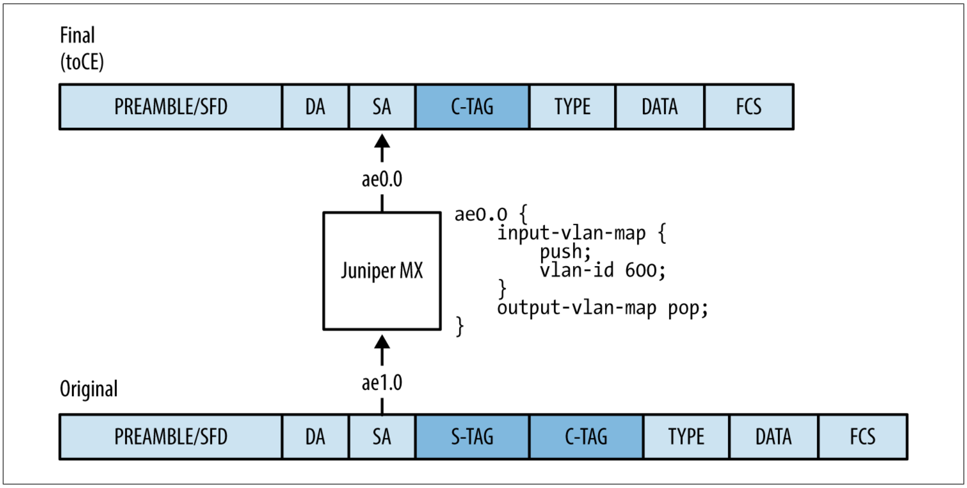
Example: Push and Pop

Let's start with one of the most commonly used stack operations: push and pop. When you want to perform IEEE 802.1QinQ and add a S-TAG, it's very easy to push an S-TAG onto an ingress frame from the customer edge (CE) and pop the S-TAG when sending egress frames destined to the CE, as shown in Figure 2-25.



```
ae0.0 {
    input-vlan-map {
        push;
        vlan-id 600;
    }
    output-vlan-map pop;
}
```

Figure 2-25. Example of ingress IEEE 802.1QinQ push operation



```
ae0.0 {
    input-vlan-map {
        push;
        vlan-id 600;
    }
    output-vlan-map pop;
}
```

Figure 2-26. Example of egress IEEE 802.1QinQ pop operation

In this example, the MX receives an ingress frame on interface `ae0.0`, which has a single C-TAG. Looking at the `input-vlan-map`, you can see that the stack operation is pushing an S-TAG with a `vlan-id` of 600. As the frame is bridged and egresses interface `ae1.0`, it now has both a S-TAG and C-TAG.

What happens when a frame egresses interface `ae0.0`? The opposite is true. The `output-vlan-map` that's associated with `ae0.0` will be applied to any egress frames. Take a closer took by studying Figure 2-26.

Notice that the original frame has both a S-TAG and C-TAG. This frame is destined back to the CE so it will have to egress the interface `ae0.0`. Because there's an `output-`

vlan-map associated with ae0.0, the frame is subject to the pop operation. This will remove the outermost frame, which is the S-TAG. When the CE receives the frame, it now only contains the original C-TAG.

Example: Swap-Push and Pop-Swap

Now let's take a look at two of the more advanced stack operations: swap-push and pop-swap. Remember that at a high level, the only purpose of input-vlan-map and output-vlan-map is to apply a consistent and reversible VLAN map to ingress and egress frames.

In this example, our requirements are:

Ingress

1. Receive IEEE 802.1Q frames with a single tag with a VLAN ID of 2.
2. Swap VLAN ID 2 with VLAN ID 4.
3. Push an outer tag with VLAN ID 5.
4. Transmit IEEE 802.1QinQ frames with an outer VLAN ID of 5 and inner VLAN ID of 4.

Egress

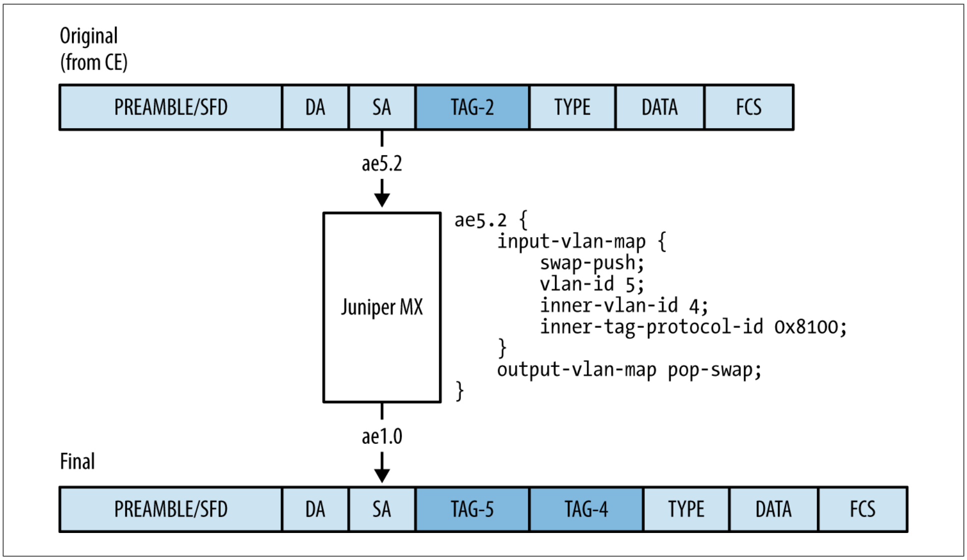
1. Receive IEEE 802.1QinQ frames with an outer VLAN ID of 5 and inner VLAN ID of 4.
2. Pop the outer tag.
3. Swap VLAN ID 4 with VLAN ID 2.
4. Transmit IEEE 802.1Q frames with a single tag with VLAN ID of 2.

Let's begin by creating the input-vlan-map function. Because you're taking an ingress frame with a single tag and turning it into a frame with a dual tag, you'll need to define the inner TPID. In this example, let's use 0x8100. You also need to specify the inner and outer VLAN IDs. The outer-vlan-map function is very easy: Simply use pop-swap without any additional arguments, and because no arguments are specified, outer-vlan-map will use the ae5.2 vlan-id as the default.

```
interfaces {
  ae5 {
    flexible-vlan-tagging;
    encapsulation flexible-ethernet-services;
    unit 2 {
      encapsulation vlan-bridge;
      vlan-id 2;
      input-vlan-map {
        swap-push;
        inner-tag-protocol-id 0x8100;
        vlan-id 5;
        inner-vlan-id 4;
      }
```

```
            output-vlan-map pop-swap;
        }
    }
}
```

Now let's take a look at how the packet is transformed as it ingresses and egresses interface ae5.2, as shown in Figure 2-27.

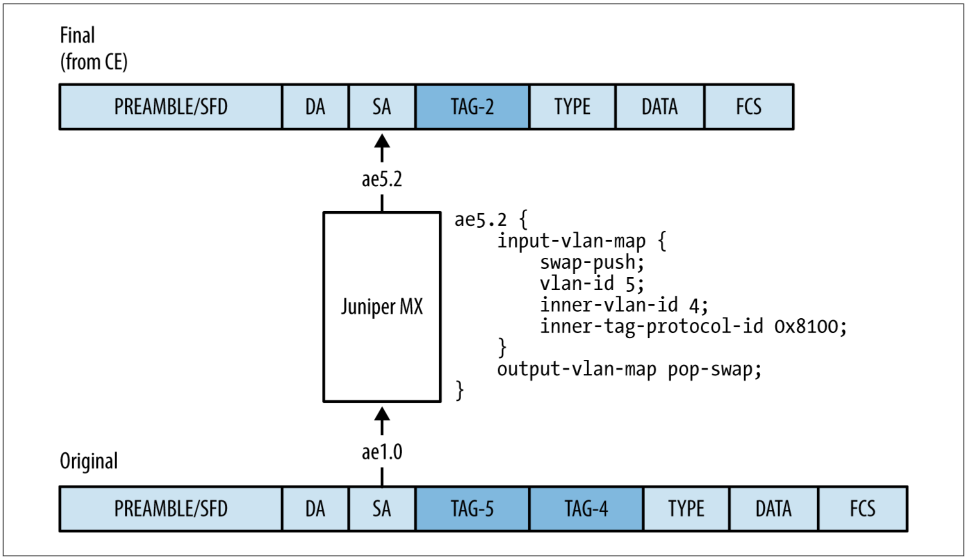


```
ae5.2 {
    input-vlan-map {
        swap-push;
        vlan-id 5;
        inner-vlan-id 4;
        inner-tag-protocol-id 0x8100;
    }
    output-vlan-map pop-swap;
}
```

Figure 2-27. Ingress swap-push

As you can see, the packet ingresses interface ae5.2, and the outer tag is swapped with VLAN ID 4. Next push VLAN ID 5, which becomes the new outer tag while VLAN ID becomes the new inner tag. Set the inner TPID to 0x8100.

Now let's review the VLAN mapping in the opposite direction in Figure 2-28.



```
ae5.2 {
    input-vlan-map {
        swap-push;
        vlan-id 5;
        inner-vlan-id 4;
        inner-tag-protocol-id 0x8100;
    }
    output-vlan-map pop-swap;
}
```

Figure 2-28. Egress pop-swap

The interface `ae5.2` receives the dual tagged frame and pops the outer tag. Next swap the remaining tag with interface `ae5.2 vlan-id` of 2.

As engineers, we always trust, but we also need to verify. Let's take a look at the interface `ae5.2` to confirm the VLAN mapping is working as expected:

```
1    dhanks@R2-RE0> show interfaces ae5.2
2      Logical interface ae5.2 (Index 345) (SNMP ifIndex 582)
3        Flags: SNMP-Traps 0x24004000
VLAN-Tag [ 0x8100.2 ] In(swap-push .5 0x0000.4) Out(pop-swap .2)
  Encapsulation: VLAN-Bridge
4        Statistics        Packets        pps         Bytes          bps
5        Bundle:
6            Input :       1293839        424      131971384       346792
7            Output:       1293838        424      131971350       346792
8        Protocol bridge, MTU: 1522
```

Notice on line 3 there are three important things: `VLAN-Tag`, `In`, and `Out`. The `VLAN-Tag` tells us exactly how the IFL is configured to receive ingress frames. The `In` and `Out` are the `input-vlan-map` and `output-vlan-map` functions. As expected, you can see that `input-vlan-map` is using `swap-push` with the VLAN IDs 5 and 4. Likewise, the `output-vlan-map` is using `pop-swap` with the IFL's `vlan-id` of 2.

Bridge Domains

Wow. Who would have guessed that bridge domains would be so far down into the chapter? There was a lot of fundamental material that needed to be covered before encountering bridge domains. First, take a look at Figure 2-29.

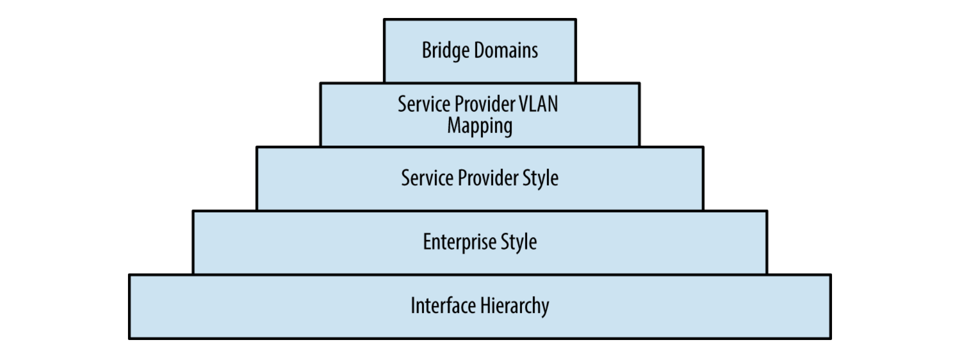


Figure 2-29. Learning hierarchy of Juniper MX bridging

Because the MX allows for advanced customization of bridging, it was best to start at the bottom and work up through interface hierarchy, configuration styles, and finally VLAN mapping.

Let's get into it. What's a bridge domain?

A *bridge domain* is simply a set of IFLs that share the same flooding, filtering, and forwarding characteristics. A bridge domain and *broadcast domain* are synonymous in definition and can be used interchangeably with each other.

Learning Domain

Bridge domains require a method to learn MAC addresses. This is done via a *learning domain*. A learning domain is simply a MAC forwarding database. Bridge domains by default have a single learning domain, but it's possible to have multiple learning domains per bridge domain, as shown in Figure 2-30.

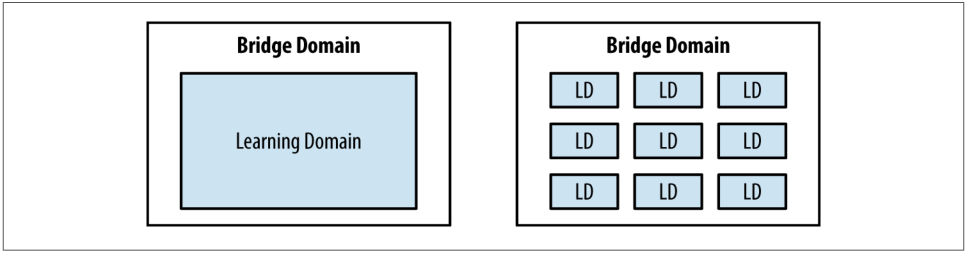


Figure 2-30. Illustration of Single Learning Domain and Multiple Learning Domains per Bridge Domain

Single Learning Domain

A single learning domain per bridge domain is the system default. When a bridge domain is configured with a single learning domain, all MAC addresses learned are associated with the learning domain attached to the bridge domain.

The most common use case is creating a standard, traditional switch with no qualified MAC learning. Within the bridge domain, only unique MAC addresses can exist as there is a single learning domain.

Let's take a look at how to create a default bridge domain with a single learning domain, using VLAN IDs of 100 and 200 and naming them appropriately:

```
dhanks@R2-RE0> show configuration bridge-domains
VLAN100 {
    vlan-id 100;
    routing-interface irb.100;
}
VLAN200 {
    vlan-id 200;
    routing-interface irb.200;
}
```

The bridge domains are created under the `bridge-domain` hierarchy, calling our bridge domains `VLAN100` and `VLAN200`. Now let's take a look at a couple of show commands and see the bridge domain to learning domain relationship:

```
dhanks@R2-RE0> show bridge domain

Routing instance        Bridge domain        VLAN ID     Interfaces
default-switch          VLAN100              100         ae0.0
                                                         ae1.0
                                                         ae2.0
default-switch          VLAN200              200         ae0.0
                                                         ae1.0
                                                         ae2.0
```

As expected, the bridge domains under the `default-switch` routing instance with a VLAN ID of 100 and 200. Let's take a look at the MAC database for this bridge domain.

```
{master}
dhanks@R2-RE0> show bridge mac-table

MAC flags (S -static MAC, D -dynamic MAC, L -locally learned
           SE -Statistics enabled, NM -Non configured MAC, R -Remote PE MAC)

Routing instance : default-switch
 Bridging domain : VLAN100, VLAN : 100
   MAC                 MAC      Logical
   address             flags    interface
   5c:5e:ab:6c:da:80   D        ae0.0
   5c:5e:ab:72:c0:80   D        ae2.0
   5c:5e:ab:72:c0:82   D        ae2.0

MAC flags (S -static MAC, D -dynamic MAC, L -locally learned
           SE -Statistics enabled, NM -Non configured MAC, R -Remote PE MAC)

Routing instance : default-switch
 Bridging domain : VLAN200, VLAN : 200
   MAC                 MAC      Logical
   address             flags    interface
   5c:5e:ab:6c:da:80   D        ae1.0
   5c:5e:ab:72:c0:80   D        ae2.0
   5c:5e:ab:72:c0:82   D        ae2.0
```

Again, you can see that there's a single learning domain attached to each bridge domain. It's interesting to note that the two learning domains contain the exact same information.

Multiple Learning Domains

Having more than one learning domain is a bit of a corner case. The primary use case for multiple learning domains is for Service Providers to tunnel customer VLANs. The big advantage is that multiple learning domains allows for qualified MAC address learning. For example, the same MAC address can exist in multiple learning domains at the same time.

Let's take a look how to configure multiple learning domains. It's very straightforward and requires a single option `vlan-id all`:

```
{master}[edit]
dhanks@R1-RE0# show bridge-domains
ALL {
    vlan-id all;
    interface ae0.100;
    interface ae0.200;
    interface ae1.100;
    interface ae1.200;
    interface ae2.100;
    interface ae2.200;
}
```

In order to create a bridge domain with multiple learning domains, use the `vlan-id all` option. Let's take a look at the MAC table for bridge domain `ALL` and look for a single bridge domain with multiple learning domains:

```
{master}
dhanks@R1-RE0> show bridge mac-table bridge-domain ALL

MAC flags (S -static MAC, D -dynamic MAC, L -locally learned
           SE -Statistics enabled, NM -Non configured MAC, R -Remote PE MAC)

Routing instance : default-switch
 Bridging domain : ALL, VLAN : 100
   MAC                 MAC      Logical
   address             flags    interface
   5c:5e:ab:6c:da:80   D        ae1.100
   5c:5e:ab:72:c0:80   D        ae2.100
```

```
MAC flags (S -static MAC, D -dynamic MAC, L -locally learned
           SE -Statistics enabled, NM -Non configured MAC, R -Remote PE MAC)

Routing instance : default-switch
 Bridging domain : ALL, VLAN : 200
   MAC                 MAC      Logical
   address             flags    interface
   5c:5e:ab:6c:da:80   D        ae0.200
   5c:5e:ab:72:c0:80   D        ae2.200
```

In this specific example, the bridge domain ALL has two learning domains. There's a learning domain for each VLAN configured on the IFLs in the bridge domain. The example is only using VLAN ID 100 and 200, so there are two learning domains.

The interesting thing to note is that you can see the qualified MAC learning in action. The MAC address 5c:5e:ab:6c:da:80 exists in both learning domains associated with different IFLs and VLANs IDs, but note there's only a single bridge domain.

Bridge Domain Modes

Bridge domains have six different modes available: default, none, all, range, single, and dual, as listed in Table 2-4. Bridge domain modes were introduced to the MX to provide shortcuts so that it's no longer required use input-vlan-map and output-vlan-map commands on every single IFL in the bridge-domain. Bridge-domain modes are the easiest way to provide VLAN normalization through automatic VLAN mapping.

Table 2-4. Bridge-Domain Modes.

Mode	Learning Domains	Bridge-Domains	Bridge-Domain Tags	IRB	Enterprise Style IFLs	Service Provider Style IFLs	VLAN Mapping	Bridge-Domain Limit Per Routing Instance
Default	1	1	N/A	No	No	Yes	Manual	Unlimited
None	1	1	0	Yes	No	Yes	Automatic	Unlimited
All	4094	1	1	No	No	Yes	Automatic	1
List	N[a]	N[a]	1	No	Yes	No	Automatic	1
Single	1	1	1	Yes	Yes	Yes	Automatic	4094
Dual	1	1	2	Yes	No	Yes	Automatic	16.7M

[a] When a bridge domain mode is set to be in a list, the number of bridge domain and learning domains depend on how many VLANs there are in the list. For example, if the list is bridge-domain vlan-id-list [1-10] there would be 10 bridge domains and 10 learning domains. The ratio of bridge domains to learning domains will always be 1:1.

Each of the bridge domain modes provides a different type of VLAN mapping automation. This makes the process of supporting Ethernet-based services seamless and simple to configure. For example, what if you had to deliver a Layer 2 network to three different locations, each requiring a different VLAN ID? By using bridge domain modes, you're able to deliver this type of service by simply configuring the IFLs as necessary and including them into a bridge domain. The bridge domain mode takes care of all calculating VLAN mapping, stack operations, and normalization.

Default

The default mode behaves just like the M/T Series and requires that you configure `input-vlan-map` and `output-vlan-map` commands on every single IFL. There is no VLAN mapping automation, and it is required that you define each stack operation manually. This mode is used when you require advanced VLAN mapping that isn't available through automatic VLAN mapping functions used with other bridge domain modes addressed later in the chapter.

None

The bridge domain mode of `none` is very interesting because it strips the frame of any tags inside of the bridge-domain. There's no requirement to use the same VLAN ID on the IFLs that make up a bridge domain mode of `none`. There's also no restriction on mixing IFLs with single, dual, or no tags at all. This can make for a very creative Layer 2 network like that shown in Figure 2-31.

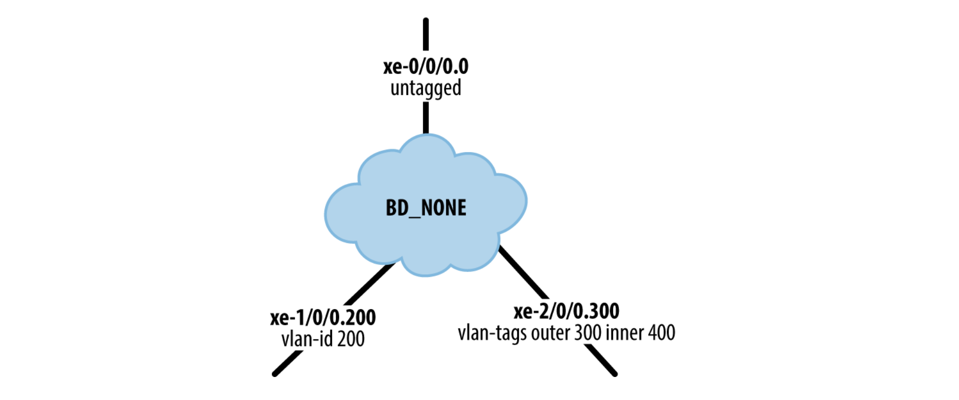


Figure 2-31. Bridge-Domain vlan-id none.

This bridge domain contains three IFLs, all of which differ in the number of tags as well as VLAN IDs. As frames are bridged across this network, each IFL will automatically pop tags so that each frame is stripped of all tags entering and leaving the bridge domain.

A bridge domain in the vlan-id none mode cannot support IFLs using the Enterprise-style interface configuration. It's required to use the Service Provider-style interface configuration.

Regardless of the VLAN ID and the number of tags on an IFL, the bridge domain mode none only has a single learning domain. When interface xe-0/0/0.0 sends Broadcast, Unknown Unicast, or Multicast (BUM), it will be sent out to every other IFL within the bridge domain. Once the traffic reaches every other IFL in the bridge domain, each IFL will perform any automatic VLAN mapping to send the traffic out of the bridge domain. Because there is a single learning domain, there can be no overlapping MAC addresses within the bridge domain.

Let's review the configuration:

```
interfaces {
    xe-0/0/0 {
        encapsulation ethernet-bridge;
        unit 0;
    }
    xe-1/0/0 {
        encapsulation flexible-ethernet-services;
        flexible-vlan-tagging;
        unit 200 {
            encapsulation vlan-bridge;
            vlan-id 200;
        }
    }
    xe-2/0/0 {
        encapsulation flexible-ethernet-services;
        flexible-vlan-tagging;
        unit 300 {
            encapsulation vlan-bridge;
            vlan-tags outer 300 inner 400;
        }
    }
}
bridge-domains {
    BD_NONE {
        vlan-id none;
        interface xe-0/0/0.0;
        interface xe-1/0/0.200;
        interface xe-2/0/0.300;
    }
}
```

As mentioned previously, bridge domain modes automatically apply VLAN mapping to the IFLs. Each IFL has its own automatic VLAN mapping depending on how it's configured, as listed in Table 2-5.

Table 2-5. Bridge-Domain Mode None Automatic VLAN Mapping.

IFL	input-vlan-map	output-vlan-map
xe-0/0/0.0	none	none
xe-1/0/0.200	pop	push
xe-2/0/0.300	pop-pop	push-push

You can verify this automatic VLAN mapping with the `show interfaces` command for each IFL. Let's take a look at `xe-1/0/0.200` and `xe-2/0/0.300`:

```
dhanks@R1-RE0> show interfaces xe-1/0/0.200
  Logical interface xe-1/0/01.200 (Index 354) (SNMP ifIndex 5560)
    Flags: SNMP-Traps 0x24004000 VLAN-Tag [ 0x8100.200 ] In(pop) Out(push 0x0000.200)
  Encapsulation: VLAN-Bridge
    Statistics        Packets        pps         Bytes          bps
    Bundle:
        Input :             4          0           244            0
        Output:             0          0             0            0
    Protocol bridge, MTU: 1522
```

You can see that `xe-1/0/0.200` is accepting ingress frames with the VLAN ID of 200. The automatic `In` function is using the `pop` operation to clear the single tag so that the frame becomes untagged as it's switched through the bridge. The automatic `Out` function takes incoming untagged frames and pushes the VLAN ID 200.

Let's verify interface `xe-2/0/0.300`:

```
dhanks@R1-RE0> show interfaces xe-2/0/0.300
  Logical interface xe-2/0/0.300 (Index 353) (SNMP ifIndex 5602)
    Flags: SNMP-Traps 0x20004000 VLAN-Tag [ 0x8100.300 0x8100.400 ] \ In(pop-pop)
             Out(push-push 0x0000.300 0x0000.400)
  Encapsulation: VLAN-Bridge
    Statistics        Packets        pps         Bytes          bps
    Bundle:
        Input :             0          0             0            0
        Output:             0          0             0            0
    Protocol bridge, MTU: 1522
```

You can see that this IFL has a bit more happening because of the dual tags. When it receives ingress frames, it needs to perform a `pop-pop` to remove both tags before being bridged. Egress frames need to have the dual tags applied with a `push-push` so the downstream device gets the appropriate tags.

All

So the opposite of bridge domain mode of `none` would be `all`, right? Not quite. The interesting thing about bridge domain mode `all` is that there's a single bridge domain but 4,094 learning domains.

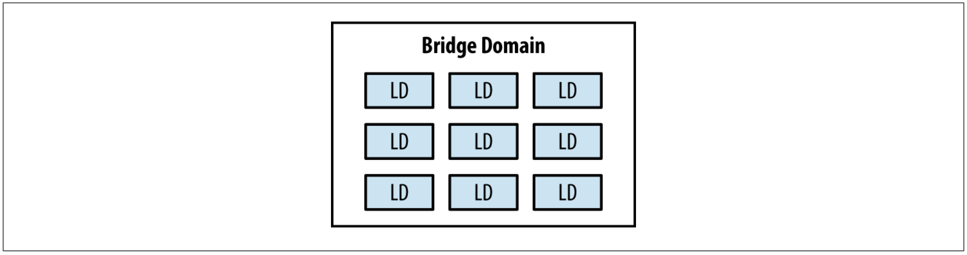


Figure 2-32. Single Bridge-Domain with Multiple Learning Domains.

The astute reader will see that the number of learning domains is mapped to each VLAN ID, enabling qualified MAC learning on a per-VLAN ID basis. This is a perfect for tunneling customer VLANs with a single command `set bridge-domain BD-ALL vlan-id all`.

A bridge domain in the `all` mode cannot support IFLs using the Enterprise-style interface configuration. It's required to use the Service Provider-style interface configuration.

The automatic VLAN mapping is a bit hybrid for the bridge domain mode `all`. Frames entering and leaving the bridge domain will have a single VLAN tag that corresponds to the `vlan-id` on an IFL with a single tag and the `vlan-tags inner` on an IFL with dual tags. So, in summary, only VLAN mapping automation happens for frames with dual tags. The outer tag is popped and uses the inner tag for bridging. In order for the frame to be switched, the egress IFL needs to have a matching VLAN ID.

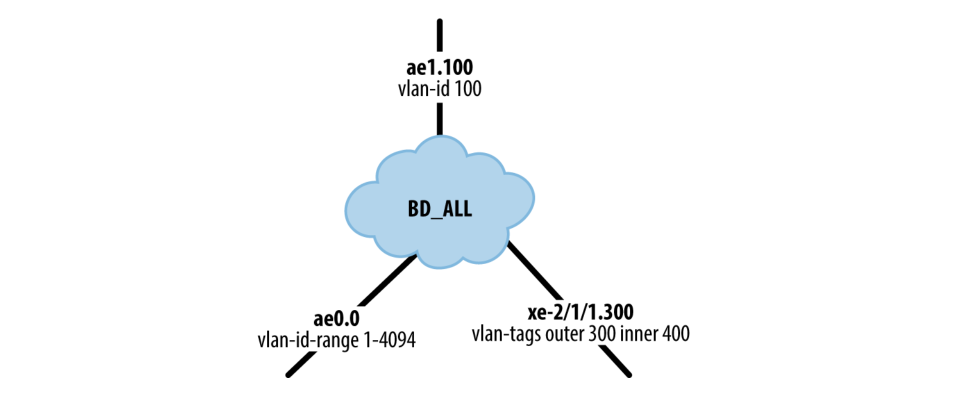


Figure 2-33. Bridge-Domain Mode vlan-id all.

Although there are 4,094 learning domains with qualified MAC learning, a single bridge domain still exists. Because of the single bridge domain, all ingress BUM frames received on a source IFL must be sent to all other destination IFLs in the `bridge-domain`, even if the destination IFL doesn't have a matching `vlan-id`. What happens is that the destination IFL receives the frame, inspects the `vlan-id`, and if there's a match it will bridge the frame. If the `vlan-id` doesn't match the destination IFL, it's simply discarded.

Let's take the example of `ae1.100` being the source IFL with a `vlan-id` of 100. When ingress BUM frames on `ae1.100` are received, they are sent to all destination IFLs, which include `ae0.0` and `xe-2/1/1.300`. IFL `ae0.0` inspects the ingress frame and sees that it has a `vlan-id` of 100. Because `ae0.0` has a `vlan-id-range 1-4094`, it considers the frame a match and accepts and bridges the frame. IFL `xe-2/1/1.300` also receives the ingress frame with a `vlan-id` of 100. Because `xe-2/1/1.300` doesn't have a matching `vlan-id`, it simply discards the frame.

The astute reader should realize that if you have `bridge-domain vlan-id all` with a lot of IFLs that do not have matching VLAN IDs or ranges, it becomes a very inefficient way to bridge traffic. Take, for example, `bridge-domain vlan-id all` with 100 IFLs. Suppose that only two of the IFLs accepted frames with a `vlan-id` of 100. What happens is that when an ingress BUM frame with a `vlan-id` of 100 enters the bridge domain, it has to be flooded to the other 99 IFLs. This is because although there are 4,094 learning domains, there's still only a single bridge domain, so the traffic has to be flooded throughout the bridge domain.

> Because `bridge-domain vlan-id all` has 4,094 learning domains and a single bridge domain, all ingress BUM frames have to be flooded to all IFLs within the bridge domain, regardless if the destination IFL can accept the frame or not.
>
> It's recommended that if you use `bridge-domain vlan-id all` you only include IFLs with matching VLAN IDs. This prevents unnecessary flooding across the SCB. For example, use matching IFL configurations such as `vlan-id-range 300-400` throughout the bridge domain.

Let's take a look at how to configure this type of bridge domain:

```
interfaces {
    xe-2/1/1 {
        flexible-vlan-tagging;
        encapsulation flexible-ethernet-services;
        unit 300 {
            encapsulation vlan-bridge;
            vlan-tags outer 300 inner 400;
        }
    }
    ae0 {
        flexible-vlan-tagging;
        encapsulation flexible-ethernet-services;
        unit 0 {
            encapsulation vlan-bridge;
            vlan-id-range 1-4094;
        }
    }
    ae1 {
        flexible-vlan-tagging;
        encapsulation flexible-ethernet-services;
        unit 100 {
            encapsulation vlan-bridge;
            vlan-id 100;
        }
    }
}
bridge-domains {
    BD_ALL {
        vlan-id all;
        interface ae1.100;
        interface ae0.0;
        interface xe-2/1/1.300;
    }
}
```

This example is interesting because of the diversity of the IFLs. For example, `xe-2/1/1.300` has dual tags, `ae0.0` has a range of tags, and `ae1.100` has a single tag. The only automatic VLAM mapping that needs to happen is with the dual tag IFL `xe-2/1/1.300`. Recall that when using `bridge-domain vlan-id all`, the VLAN ID corresponds to the VLAN ID on each IFL, except when IFLs with two tags. In the case with an IFL with dual tags, you simply pop the outer label and bridge the inner. Let's take a closer look at each interface:

```
{master}
dhanks@R1-RE0> show interfaces xe-2/1/1.300
  Logical interface xe-2/1/1.300 (Index 345) (SNMP ifIndex 5605)
    Flags: SNMP-Traps 0x20004000
VLAN-Tag [ 0x8100.300 0x8100.400 ] In(pop) Out(push 0x8100.300)
  Encapsulation: VLAN-Bridge
    Input packets : 0
    Output packets: 123
    Protocol bridge, MTU: 1522
```

Notice that the `xe-2/1/1.300` has two tags, but ingress frames are subject to `pop` and egress frames are subject to a `push`.

```
{master}
dhanks@R1-RE0> show interfaces ae1.100
  Logical interface ae1.100 (Index 343) (SNMP ifIndex 5561)
    Flags: SNMP-Traps 0x20004000 VLAN-Tag [ 0x8100.100 ]  Encapsulation: VLAN-Bridge
    Statistics        Packets        pps         Bytes          bps
    Bundle:
        Input :             2          0           124            0
        Output:           292          0         19844          272
    Protocol bridge, MTU: 1522
```

There's no automatic VLAN mapping for IFL `ae1.100`. As you can see, it's simply `0x8100.100`.

```
{master}
dhanks@R1-RE0> show interfaces ae0.0
  Logical interface ae0.0 (Index 346) (SNMP ifIndex 5469)
    Flags: SNMP-Traps 0x20004000 VLAN-Tag [ 0x8100.1-4094 ]  Encapsulation: VLAN-Bridge
    Statistics        Packets        pps         Bytes          bps
    Bundle:
        Input :          7921          1        539414          544
        Output:            86          0          6662            0
    Protocol bridge, MTU: 1522
```

Finally, with IFL `ae0.0`, you can see that it's also not performing any automatic VLAN mapping. It simply bridges all VLAN IDs between 1 and 4094.

> The IFL option `vlan-id-range` can only be used with `bridge-domain vlan-id all`.

It's an interesting thing—a single bridge domain with 4,094 learning domains. How can this be verified? The best method is to view the MAC table of the bridge domain using the `show bridge mac-table` command:

```
{master}
dhanks@R1-RE0> show bridge mac-table

MAC flags (S -static MAC, D -dynamic MAC, L -locally learned
           SE -Statistics enabled, NM -Non configured MAC, R -Remote PE MAC)

Routing instance : default-switch
 Bridging domain : BD_ALL, VLAN : 100
   MAC                 MAC      Logical
   address             flags    interface
   5c:5e:ab:6c:da:80   D        ae1.100
   5c:5e:ab:72:c0:80   D        ae0.0
   5c:5e:ab:72:c0:82   D        ae0.0

MAC flags (S -static MAC, D -dynamic MAC, L -locally learned
           SE -Statistics enabled, NM -Non configured MAC, R -Remote PE MAC)

Routing instance : default-switch
 Bridging domain : BD_ALL, VLAN : 200
   MAC                 MAC      Logical
   address             flags    interface
   5c:5e:ab:72:c0:82   D        ae0.0
```

Notice the bridge-domain `BD_ALL` has learning domains active in this example. In the authors' test bed, there is traffic going across VLAN ID 100 and 200, thus there are two active learning domains. That qualified MAC learning is taking place can also be verified because the MAC address `5c:5e:ab:72:c0:82` exists in both learning domains.

List

This bridge mode is the most recent mode added to the family of bridge domain modes and can be functionally identical to `bridge-domain vlan-id all`. The major difference is that `bridge-domain vlan-id-list` creates a bridge domain and learning domain for every VLAN specified. For example, `bridge-domain vlan-id-list 1-10` would create 10 bridge domains and 10 learning domains, whereas `bridge-domain vlan-id-list 1-4094` would create 4,094 bridge domains and 4,094 learning domains. The ratio of bridge domains to learning domains is always 1:1, as shown in Figure 2-34.

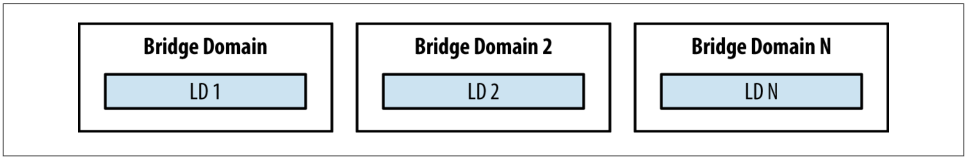


Figure 2-34. Bridge-Domain Mode List: Bridge-Domain to Learning Domain Ratio is Always 1:1.

The new `bridge-domain vlan-id-list` mode was created to complement the `bridge-domain vlan-id all` mode. The key benefit is that you no longer have to worry about making sure that the IFL's `vlan-id` is matched up to prevent unnecessary flooding across the SCB. When you create `bridge-domain vlan-id-list`, Junos treats this as a shortcut to create *N* bridge domains and learning domains. Let's take a look at an example:

```
interfaces {
    xe-2/1/1 {
        unit 0 {
            family bridge {
                interface-mode trunk;
                vlan-id-list 300;
            }
        }
    }
    ae0 {
        unit 0 {
            family bridge {
                interface-mode trunk;
                vlan-id-list 1-4094;
            }
        }
    }
    ae1 {
        unit 0 {
            family bridge {
                interface-mode trunk;
                vlan-id-list 1-4094;
            }
        }
    }
}
bridge-domains {
    BD_LIST {
        vlan-id-list 1-4094;
    }
}
```

Notice that when creating a `bridge-domain vlan-id-list`, there isn't a list of interfaces. This is because `bridge-domain vlan-id-list` requires to you use Enterprise-style interface configurations. When you `commit` the configuration, Junos automatically walks the interface structure, finds all of the IFLs, and matches them to the corresponding bridge domains.

Service Provider-style interface configurations aren't supported when using `bridge-domain vlan-id-list`.

The `vlan-id-list` is really just a shortcut to create *N* bridge domains. Let's take a closer look with `show bridge domain`:

```
{master}
dhanks@R1-RE0> show bridge domain

Routing instance        Bridge domain          VLAN ID     Interfaces
default-switch          BD_LIST-vlan-0001      1           ae0.0
                                                           ae1.0
default-switch          BD_LIST-vlan-0002      2           ae0.0
                                                           ae1.0
default-switch          BD_LIST-vlan-0003      3           ae0.0
                                                           ae1.0
default-switch          BD_LIST-vlan-0004      4           ae0.0
                                                           ae1.0
```

Notice that the bridge domain `BD_LIST` is named, but the name that appears in `show bridge-domain` has the VLAN ID appended in the format of `-vlan-N`. This eases the creation of 4,094 bridge domains, using the Enterprise-style interface configuration, and doesn't waste switch fabric bandwidth when flooding across the bridge domain.

There can only be one `bridge-domain vlan-id-list` per routing instance.

Recall that `bridge-domain vlan-id all` is a single bridge domain, and that any BUM traffic has to be flooded out to all IFLs in the bridge domain, regardless if the destination IFL can process the frame or not, as shown in Figure 2-35.

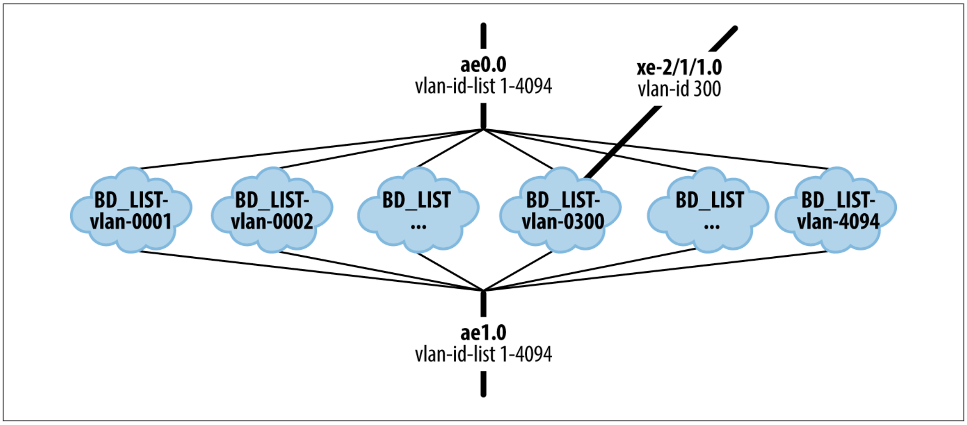


Figure 2-35. Illustration of BD_LIST Example with Three IFLs.

When using `bridge-domain vlan-id-list`, Junos creates a bridge domain for each VLAN ID. Flooding BUM traffic will only happen in the constraints of that bridge domain. Because Junos automatically places IFLs into the corresponding bridge domain, it is guaranteed that the destination IFL will be able to process any frames it receives.

> Remember that automatic VLAN mapping isn't performed on bridge domain mode `vlan-id-list` because it requires the use of Enterprise-style interface configuration.

Because `bridge-domain vlan-id-list` is simply a shortcut to create *N* bridge domains, you'll see each bridge domain listed out individually with `show bridge-domain`. If you wish to set some custom settings on a specific VLAN ID, you'll need to pull that VLAN ID out of the `vlan-id-list` and explicitly configure it by hand. Let's take a look:

```
bridge-domains {
    BD_LIST {
        vlan-id-list [ 100 200-249 251-299 ];
    }
    BD_LIST-vlan-0250 {
        vlan-id 250;
        bridge-options {
            interface-mac-limit {
                2000;
            }
        }
    }
}
```

In this example, the bridge domain `BD_LIST` supports a variety of VLAN IDs. Suppose you wanted to have a bridge domain for VLAN ID 250 but wanted to increase the default MAC limit per interface for VLAN ID 250. In order to do this, you have to remove VLAN ID 250 from the BD_LIST and place it into its own bridge domain. There's no requirement for the bridge domain name, but the author decided to use BD_LIST-vlan-0250 to keep with the vlan-id-list naming convention.

The bridge domain BD_LIST-vlan-0250 interface MAC limit has been increased to 2,000. Let's verify with show l2-learning interface:

```
{master}
dhanks@R1-RE0> show l2-learning interface
Routing Instance Name : default-switch
Logical Interface flags (DL -disable learning, AD -packet action drop,
                         LH - MAC limit hit, DN - Interface Down )
Logical             BD         MAC        STP          Logical
Interface           Name       Limit      State        Interface flags
ae0.0                          8192
                    BD_LIS..   1024       Forwarding
                    BD_LIS..   1024       Forwarding
Routing Instance Name : default-switch
Logical Interface flags (DL -disable learning, AD -packet action drop,
                         LH - MAC limit hit, DN - Interface Down )
Logical             BD         MAC        STP          Logical
Interface           Name       Limit      State        Interface flags
ae1.0                          8192
                    BD_LIS..   1024       Forwarding
                    BD_LIS..   1024       Forwarding
Routing Instance Name : default-switch
Logical Interface flags (DL -disable learning, AD -packet action drop,
                         LH - MAC limit hit, DN - Interface Down )
Logical             BD         MAC        STP          Logical
Interface           Name       Limit      State        Interface flags
xe-2/1/1.0                     8192
                    BD_LIS..   2000       Forwarding
```

It's a bit difficult to see because the BD column was truncated, but the bridge domain BD_LIST-vlan-0250 is showing a MAC limit of 2,000 as opposed to 1,024 as the other bridge domains.

Single

A bridge domain with a single VLAN ID is the most basic bridge domain possible and is what you typically see in Enterprise environments. You simply create a bridge domain, assign a VLAN ID to it, throw in a few IFLs, and call it a day. The astute reader should already know how to configure such a basic bridge domain at this point in the chapter:

```
interfaces {
    xe-2/1/1 {
        flexible-vlan-tagging;
        encapsulation flexible-ethernet-services;
        unit 300 {
            vlan-tags outer 300 inner 400;
        }
    }
    ae0 {
        unit 0 {
            family bridge {
                interface-mode trunk;
                vlan-id-list [ 100 200 ];
            }
        }
    }
    ae1 {
        flexible-vlan-tagging;
        encapsulation flexible-ethernet-services;
        unit 200 {
            encapsulation vlan-bridge;
            vlan-id 200;
        }
        unit 1000 {
            encapsulation vlan-bridge;
            vlan-id 1000;
        }
    }
}
bridge-domains {
    BD100 {
        vlan-id 100;
        interface ae1.200;
        interface ae1.1000;
        interface xe-2/1/1.300;
    }
    BD200 {
        vlan-id 200;
    }
}
```

We have to keep you on your toes and mix Enterprise-style and Service Provider-style interface configuration as well as including single and dual tags on IFLs! As you can see, a single bridge domain is pretty flexible. It can simply bridge a VLAN ID or perform automatic VLAN mapping when combined with Service Provider-style interface configuration, as shown in Figure 2-36.

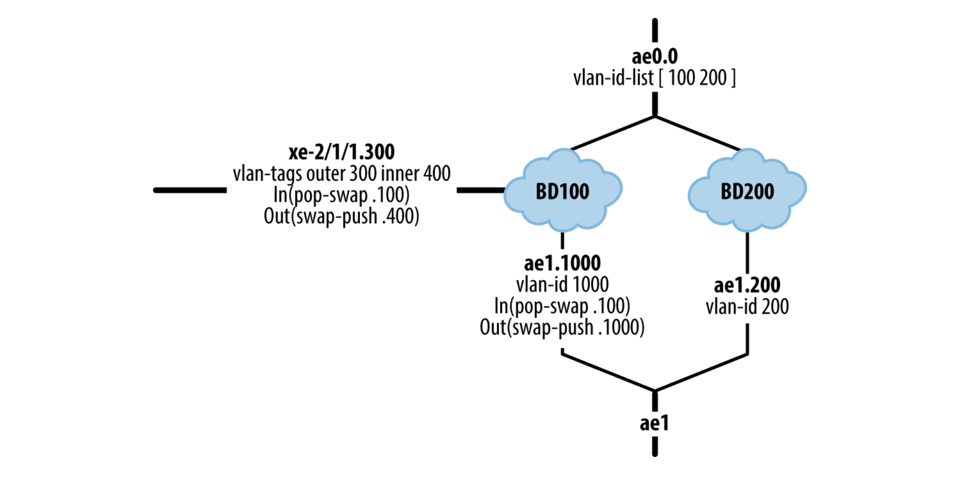


Figure 2-36. Pair of Single Bridge-Domains with a Mixture of Enterprise-style and Service Provider–style Interface Configuration.

The interesting concept with this example is that we're using a pair of single bridge domains—`BD100` and `BD200`—but each bridge domain has different types of IFL configurations and automatic VLAN mapping.

`BD100` is the most interesting bridge domain because there are many different things happening. To start, it is supporting `xe-2/1/1.300`, which is using a Service Provider-style interface configuration with dual tags. Because the outer tag of IFL `xe-2/1/1.300` is VLAN ID 300, the bridge domain has automatically applied a `pop-swap` and `swap-push` for frames entering and leaving the IFL. This maintains the C-TAG of 400 while changing the S-TAG from 300 to 100:

```
{master}
dhanks@R1-RE0> show interfaces xe-2/1/1.300
  Logical interface xe-2/1/1.300 (Index 326) (SNMP ifIndex 5605)
   Flags: SNMP-Traps 0x20004000 VLAN-Tag [ 0x8100.300 0x8100.400 ] In(pop-swap .100)
     Out(swap-push 0x8100.300 .400)  Encapsulation: VLAN-Bridge
   Input packets : 0
   Output packets: 16
   Protocol bridge, MTU: 1522
```

The other IFL using the Service Provider-style interface configuration is `ae1.1000`. This IFL is using a single tag with a `vlan-id` of 1,000. Because `BD100` has a `vlan-id` of 100, IFL `ae1.1000` performed automatic VLAN mapping to swap the two VLAN IDs as the frames enter and exit the IFL:

```
{master}
dhanks@R1-RE0> show interfaces ae1.1000
  Logical interface ae1.1000 (Index 343) (SNMP ifIndex 5606)
   Flags: SNMP-Traps 0x20004000 VLAN-Tag [ 0x8100.1000 ] In(swap .100) Out(swap .1000)
   Encapsulation: VLAN-Bridge
   Statistics        Packets        pps         Bytes          bps
```

```
    Bundle:
        Input :             4          0            312           0
        Output:            93          0           6392         272
    Protocol bridge, MTU: 1522
```

BD200 is the other bridge domain in this example. It ties together the two IFLs: ae0.0 and ae1.200. These IFLs already have support for VLAN ID 200, so there's no need to perform any automatic VLAN mapping.

Dual

A dual bridge domain is very similar to a single bridge domain except that frames entering and leaving the bridge domain have dual tags, as shown in Figure 2-37. Otherwise, these two bridge domains operate in the same fashion. Because a dual tagged bridge domain offers advanced bridging, you must use the Service Provider-style interface configuration—the Enterprise-style interface configuration isn't supported.

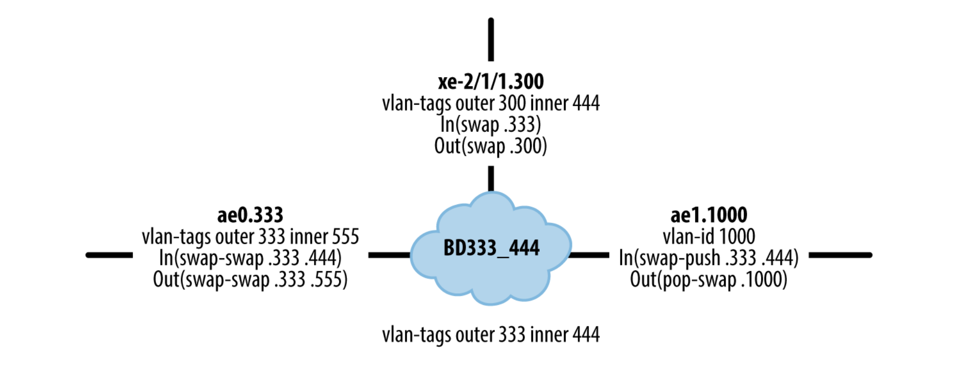


Figure 2-37. Illustration of a Bridge Domain with Dual Tags.

This is another interesting example because each of the IFLs has very specific vlan-tags that interact and conflict with the bridge-domain vlan-tags. Notice that IFL ae0.333 has the same S-TAG as the bridge domain and IFL xe-2/1/1.300 has the same C-TAG as the bridge domain:

```
interfaces {
    xe-2/1/1 {
        flexible-vlan-tagging;
        encapsulation flexible-ethernet-services;
        unit 300 {
            encapsulation vlan-bridge;
            vlan-tags outer 300 inner 444;
        }
    }
    ae0 {
        flexible-vlan-tagging;
        encapsulation flexible-ethernet-services;
        unit 333 {
            encapsulation vlan-bridge;
            vlan-tags outer 333 inner 555;
        }
    }
    ae1 {
        flexible-vlan-tagging;
        encapsulation flexible-ethernet-services;
        unit 1000 {
            encapsulation vlan-bridge;
            vlan-id 1000;
        }
    }
}
bridge-domains {
    BD333_444 {
        vlan-tags outer 333 inner 444;
        interface ae1.1000;
        interface xe-2/1/1.300;
        interface ae0.333;
    }
}
```

Interface ae0.333 has the same S-TAG as the bridge domain: 333. How does Junos simply change only the C-TAG? Because there's no such thing as null-swap, Junos just uses swap-swap instead. When the swap-swap operation is executed, Junos is smart enough to see that the outer tags match and can simply move to the next tag:

```
{master}
dhanks@R1-RE0> show interfaces ae0.333
  Logical interface ae0.333 (Index 350) (SNMP ifIndex 5609)
    Flags: SNMP-Traps 0x20004000 VLAN-Tag [ 0x8100.333 0x8100.555 ] In(swap-swap .333
      .444) Out(swap-swap .333 .555)  Encapsulation: VLAN-Bridge
    Statistics        Packets        pps         Bytes          bps
    Bundle:
        Input :             0          0             0            0
        Output:            90          1          9540          848
    Protocol bridge, MTU: 1522
```

You can see that the ingress frames are subject to swap-swap .333 .444, with the end result being that the C-TAG is swapped to 444 while the S-TAG remains as 333. When frames egress the IFL, Junos simply reverses the operation to restore the original C-TAG with swap-swap .333 .555, so that only the C-TAG is ever modified regardless of ingress or egress.

Interface xe-2/1/1.300 has the opposite configuration of ae0.333—its C-TAG is the same as the bridge domain BD333_444. The automatic VLAN mapping is very simple; Junos only needs to swap the outer tag:

```
{master}
dhanks@R1-RE0> show interfaces xe-2/1/1.300
  Logical interface xe-2/1/1.300 (Index 328) (SNMP ifIndex 5605)
    Flags: SNMP-Traps 0x20004000 VLAN-Tag [ 0x8100.300 0x8100.444 ] In(swap .333)
      Out(swap .300) Encapsulation: VLAN-Bridge
    Input packets : 0
```

```
        Output packets: 4
        Protocol bridge, MTU: 1522
```

The last interface, ae1.1000, is the simplest; it's configured only with a single tag of vlan-id 1000. As ingress frames are received, Junos will automatically swap-push .333 .444 so that the original C-TAG is changed and a new S-TAG is pushed onto the frame, resulting in a S-TAG of 333 and a C-TAG of 444:

```
{master}
dhanks@R1-RE0> show interfaces ae1.1000
  Logical interface ae1.1000 (Index 345) (SNMP ifIndex 5606)
    Flags: SNMP-Traps 0x20004000 VLAN-Tag [ 0x8100.1000 ] In(swap-push 0x0000.333 .444)
      Out(pop-swap .1000)  Encapsulation: VLAN-Bridge
    Statistics        Packets        pps         Bytes          bps
    Bundle:
        Input :          3570          1        258850          816
        Output:           195          0         17570            0
    Protocol bridge, MTU: 1522
```

The opposite is true for egress frames going through ae1.1000. The pop-swap .1000 is applied to egress frames, which pops the S-TAG of 333 and swaps the remaining tag with the original value of 1,000.

Bridge Domain Options

As you begin configuring and using the MX in your network, there will be times when you need to change the MAC learning characteristics or set limits within the routing instance or bridge domain. Let's review the most common bridge domain options.

MAC Table Size

The MX gives you several different options for how to limit the number of MAC addresses. You can do this globally across the entire system, for an entire routing instance, for a specific bridge domain, or enforce a MAC limit per interface.

Global. When the MAC limit is set globally, it takes into account all logical systems, routing instances, bridge domains, and interfaces. To set this limit, you'll need to modify protocols l2-learning:

```
protocols {
    l2-learning {
        global-mac-limit {
            100;
        }
    }
}
```

This isn't very useful, but it makes a point. Now the global MAC limit is set to 100. Let's verify this:

```
{master}
dhanks@R1-RE0> show l2-learning global-information
Global Configuration:
```

```
MAC aging interval    : 300
MAC learning          : Enabled
MAC statistics        : Enabled
MAC limit Count       : 100
MAC limit hit         : Disabled
MAC packet action drop: Disabled
LE  aging time        : 1200
LE  BD aging time     : 1200
```

It's generally a good idea to set the global MAC limit as a last resort or safety net. A common model for MAC limits is to apply the *Russian doll architecture*—start per interface and make your way up to the global limit, increasing the value as you go.

Bridge domain. Setting the MAC limit per bridge domain is very easy. First, let's set the `bridge-options`:

```
bridge-domains {
    BD333_444 {
        vlan-tags outer 333 inner 444;
        interface ae1.1000;
        interface xe-2/1/1.300;
        interface ae0.333;
        bridge-options {
            mac-table-size {
                2000;
            }
        }
    }
}
```

Now, the bridge domain `BD333_444` has a MAC limit of 2,000. Let's verify:

```
{master}
dhanks@R1-RE0> show l2-learning instance bridge-domain BD333_444 detail

Information for routing instance and bridge-domain:
Routing instance : default-switch
 Bridging domain : BD333_444
   RTB Index: 4                    BD Index: 8534
   MAC limit: 2000                 MACs learned: 3
   Sequence number: 2              Handle: 0x88b3600
   BD Vlan Id: 333,444
   Flags: Statistics enabled
   Config BD Vlan Id          : 333,444      Config operation: none
   Config params: mac tbl sz: 2000, mac age: 300000000, intf mac limit: 1024,
   Config flags: mac stats enablevlan
   Config BD Static MAC count : 0
   Config ownership flags: config
   Config RG Id: 0                           Active RG Id: 0
   Config Service Id: 0            Active Service Id: 0
   Kernel ownership flags: config
   MVRP ref count: 0
 Counters:
   Kernel write errors        : 0
```

The hidden option detail had to be used, but it got the job done. You can see that the MAC limit is now 2,000.

Interface. The last option is to limit the number of MAC addresses per IFL. This setting is applied under bridge-domains bridge-options:

```
bridge-domains {
    BD333_444 {
        vlan-tags outer 333 inner 444;
        interface ae1.1000;
        interface xe-2/1/1.300;
        interface ae0.333;
        bridge-options {
            interface-mac-limit {
                100;
            }
        }
    }
}
```

This will set a MAC limit of 100 for every IFL in the bridge domain BD333_444: ae1.1000, xe-2/1/1.300, and ae0.333:

```
{master}
dhanks@R1-RE0> show l2-learning interface
Routing Instance Name : default-switch
Logical Interface flags (DL -disable learning, AD -packet action drop,
                         LH - MAC limit hit, DN - Interface Down )
Logical             BD         MAC        STP          Logical
Interface           Name       Limit      State        Interface flags
xe-2/1/1.300                   0
                    BD333_..   100        Forwarding
Routing Instance Name : default-switch
Logical Interface flags (DL -disable learning, AD -packet action drop,
                         LH - MAC limit hit, DN - Interface Down )
Logical             BD         MAC        STP          Logical
Interface           Name       Limit      State        Interface flags
ae0.333                        0
                    BD333_..   100        Forwarding
Routing Instance Name : default-switch
Logical Interface flags (DL -disable learning, AD -packet action drop,
                         LH - MAC limit hit, DN - Interface Down )
Logical             BD         MAC        STP          Logical
Interface           Name       Limit      State        Interface flags
ae1.1000                       0
                    BD333_..   100        Forwarding
```

No need for a hidden command this time around, as you can clearly see the MAC limit of 100 is now applied for all of the IFLs in the bridge domain BD333_444.

No MAC learning

Someone engineers might say, “What good is a bridge domain without a MAC table?” but there will always be a reason to disable MAC learning within a bridge domain. For example, suppose the bridge domain was tunneling customer or "customer of customer" traffic between two IFLs. At a high level, all the MX is doing is moving frames from IFL A to IFL B, so why does it care about MAC learning? But in this use case, MAC learning isn't required because there's only one IFL to which the frame could be flooded to. Let's look at the bridge domain MAC table before any changes are made:

```
dhanks@R1-RE0> show bridge mac-table

MAC flags (S -static MAC, D -dynamic MAC, L -locally learned
           SE -Statistics enabled, NM -Non configured MAC, R -Remote PE MAC)

Routing instance : default-switch
 Bridging domain : BD333_444, VLAN : 333,444
   MAC                 MAC      Logical
   address             flags    interface
   5c:5e:ab:6c:da:80   D,SE     ae1.1000
   5c:5e:ab:72:c0:80   D,SE     ae0.333
   5c:5e:ab:72:c0:82   D,SE     ae0.333
```

Looks about right. There are three MAC addresses in the bridge domain BD333_444. Now let's enable no-mac-learning:

```
bridge-domains {
    BD333_444 {
        vlan-tags outer 333 inner 444;
        interface ae1.1000;
        interface ae0.333;
        bridge-options {
            no-mac-learning;
        }
    }
}
```

Now, let's take another look at the MAC table for bridge domain BD333_444:

```
{master}
dhanks@R1-RE0> show bridge mac-table

{master}
dhanks@R1-RE0>
```

At this point, the bridge domain has turned itself into a functional hub (without all of the collision domains). It doesn't care about learning MAC addresses, and all ingress frames will be flooded to all IFLs in the bridge domain. Don't forget the golden rule of bridging loop prevention: the frame will be flooded out all interfaces except the interface from the frame was originally received.

The benefit of no-mac-learning with only two IFLs in a bridge domain is that the MX doesn't have to worry about learning MAC addresses and storing them. Imagine a scenario where you needed to provide basic Ethernet bridging between two customers and the number of MAC addresses going across the bridge domain was in the millions. With no-mac-learning, the Juniper MX simply bridges the traffic and doesn't care if there are a trillion MAC addresses, as the frames are just replicated to the other IFLs blindly. Using no-mac-learning in a bridge domain with more than three IFLs isn't recommended because it's inefficient.

Show Bridge Domain Commands

Throughout this chapter, you have become an expert at configuring and understanding the different types of bridge domains. Now let's review some of the show commands that will help you troubleshoot and verify settings related to bridging. For your reference, here is the current bridge domain that is used when demonstrating the various show commands:

```
bridge-domains {
    BD333_444 {
        vlan-tags outer 333 inner 444;
        interface ae1.1000;
        interface xe-2/1/1.300;
        interface ae0.333;
    }
}
```

It's just a single bridge domain using dual tags and contains three IFLs.

show bridge domain

Let's start with the most basic command to view the bridge domains currently configured on the system:

```
{master}
dhanks@R1-RE0> show bridge domain

Routing instance        Bridge domain          VLAN ID     Interfaces
default-switch          BD333_444              333,444
                                                           ae0.333
                                                           ae1.1000
                                                           xe-2/1/1.300
```

There are four important fields here: Routing instance, Bridge domain, VLAN ID, and Interfaces. As you configure bridge domains, it's important to verify that Junos is executing what you think you configured. In this example, you can see that the bridge domain BD333_444 is located in the default-switch, which could also be thought of as the default routing table. To the right of the bridge domain, you can see what VLAN IDs are configured and what interfaces are currently participating within the bridge domain.

show bridge mac-table

To view the bridge domain's MAC table, use the show bridge mac-table command:

```
{master}
dhanks@R1-RE0> show bridge mac-table bridge-domain BD333_444

MAC flags (S -static MAC, D -dynamic MAC, L -locally learned
           SE -Statistics enabled, NM -Non configured MAC, R -Remote PE MAC)

Routing instance : default-switch
 Bridging domain : BD333_444, VLAN : 333,444
   MAC                 MAC      Logical
   address             flags    interface
   5c:5e:ab:6c:da:80   D        ae1.1000
   5c:5e:ab:72:c0:80   D        ae0.333
   5c:5e:ab:72:c0:82   D        ae0.333
```

This example shows that there are three dynamically learned MAC addresses in the BD333_444 bridge domain. Two of the MAC addresses were learned by ae0.333, and the other MAC address was learned by ae1.1000.

show bridge statistics

A great way to get a bird's eye view of a bridge domain is to look at the statistics. From this vantage point, you're able to quickly see how many packets have been bridged.

```
{master}
dhanks@R1-RE0> show bridge statistics bridge-domain BD333_444
   Local interface: ae0.333, Index: 343
     Broadcast packets:                    2
     Broadcast bytes  :                  120
     Multicast packets:                 1367
     Multicast bytes  :                92956
     Flooded packets  :                  486
     Flooded bytes    :                49572
     Unicast packets  :                 2593
     Unicast bytes    :               264444
     Current MAC count:                    2 (Limit 1024)
   Local interface: xe-2/1/1.300, Index: 328
     Broadcast packets:                    0
     Broadcast bytes  :                    0
     Multicast packets:                    0
     Multicast bytes  :                    0
     Flooded packets  :                    0
     Flooded bytes    :                    0
     Unicast packets  :                    0
     Unicast bytes    :                    0
     Current MAC count:                    0 (Limit 1024)
   Local interface: ae1.1000, Index: 345
     Broadcast packets:                16537
     Broadcast bytes  :               992220
     Multicast packets:                    0
     Multicast bytes  :                    0
     Flooded packets  :                 2402
     Flooded bytes    :               244886
     Unicast packets  :                 4634
     Unicast bytes    :               472508
     Current MAC count:                    1 (Limit 1024)
```

As you can see, the statistics are broken out per IFL, as well as the type of packet.

show l2-learning instance detail

Another method of seeing a bird's eye view of a bridge domain is to use the l2-learning show commands. This will only show the Layer 2 learning details of the bridge domain:

```
{master}
dhanks@R1-RE0> show l2-learning instance detail

Information for routing instance and bridge-domain:
Routing instance : default-switch
 Bridging domain : BD333_444
   RTB Index: 4                    BD Index: 8534
   MAC limit: 2000                 MACs learned: 3
   Sequence number: 6              Handle: 0x88b3600
   BD Vlan Id: 333,444
   Config BD Vlan Id          : 333,444      Config operation: none
   Config params: mac tbl sz: 2000, mac age: 300000000, intf mac limit: 1024,
   Config flags: vlan
   Config BD Static MAC count : 0
   Config ownership flags: config
   Config RG Id: 0                           Active RG Id: 0
   Config Service Id: 0            Active Service Id: 0
   Kernel ownership flags: config
   MVRP ref count: 0
 Counters:
   Kernel write errors        : 0
```

This example is showing the bridge-domain BD333_444. You can see that the current MAC limit for this bridge domain is 2,000, and there have only been three MAC addresses learned thus far. You can also see the default per interface MAC limit is currently set to 1,024.

Clear MAC Addresses

Sooner or later, you're going to need to clear a MAC address entry in the bridge domain. There are several options of clearing MAC addresses: you can simply clear the entire table, or cherry-pick a specific MAC address.

Specific MAC Address

First, here's how to clear a specific MAC address:

```
{master}
dhanks@R1-RE0> show bridge mac-table

MAC flags (S -static MAC, D -dynamic MAC, L -locally learned
           SE -Statistics enabled, NM -Non configured MAC, R -Remote PE MAC)

Routing instance : default-switch
 Bridging domain : BD333_444, VLAN : 333,444
   MAC                 MAC      Logical
   address             flags    interface
   5c:5e:ab:6c:da:80   D,SE     ae1.1000
```

```
      5c:5e:ab:72:c0:80   D,SE     ae0.333
      5c:5e:ab:72:c0:82   D,SE     ae0.333
```

Suppose that you needed to clear the MAC address `5c:5e:ab:6c:da:80`. You can do so using the `clear bridge` command:

```
{master}
dhanks@R1-RE0> clear bridge mac-table bridge-domain BD333_444 5c:5e:ab:6c:da:80
```

Again, you should always verify. Let's use the `show bridge mac-table` command once again to make sure it's been removed:

```
{master}
dhanks@R1-RE0> show bridge mac-table

MAC flags (S -static MAC, D -dynamic MAC, L -locally learned
           SE -Statistics enabled, NM -Non configured MAC, R -Remote PE MAC)

Routing instance : default-switch
 Bridging domain : BD333_444, VLAN : 333,444
   MAC                 MAC      Logical
   address             flags    interface
   5c:5e:ab:72:c0:80   D,SE     ae0.333
   5c:5e:ab:72:c0:82   D,SE     ae0.333
```

As suspected, the MAC address `5c:5e:ab:6c:da:80` has been removed and is no longer in the MAC table.

Entire Bridge-Domain

You can use the same command, minus the MAC address, to completely clear the MAC table for a specific bridge domain, and dynamically remove all learned MAC addresses. Let's view the MAC table before blowing it away:

```
{master}
dhanks@R1-RE0> show bridge mac-table

MAC flags (S -static MAC, D -dynamic MAC, L -locally learned
           SE -Statistics enabled, NM -Non configured MAC, R -Remote PE MAC)

Routing instance : default-switch
 Bridging domain : BD333_444, VLAN : 333,444
   MAC                 MAC      Logical
   address             flags    interface
   5c:5e:ab:6c:da:80   D,SE     ae1.1000
   5c:5e:ab:72:c0:80   D,SE     ae0.333
   5c:5e:ab:72:c0:82   D,SE     ae0.333
```

As you can see, the bridge domain `BD333_444` has three MAC addresses. Now let's blow them all away:

```
{master}
dhanks@R1-RE0> clear bridge mac-table bridge-domain BD333_444
```

Ah. With great power comes great responsibility. Let's verify once again:

```
{master}
dhanks@R1-RE0> show bridge mac-table

{master}
dhanks@R1-RE0>
```

That definitely did it. Not a single MAC address left. If you're feeling content with this new power, let's move on to MAC accounting before we get into more trouble.

MAC Accounting

How do you know how many times a specific MAC address has been used as a source or destination when bridging? By default, this information is available and requires the use of MAC accounting. With MAC accounting enabled, Junos will enable counters in the Forwarding Information Base (FIB) to keep track of how many times each destination MAC address has been used as a source or destination when bridged. MAC accounting can be turned on globally or within a specific bridge domain. Let's take a look at the global configuration:

```
protocols {
    l2-learning {
        global-mac-statistics;
    }
}
```

Again, let's trust that this configuration is correct, but verify with the `show` command:

```
{master}
dhanks@R1-RE0> show l2-learning global-information
Global Configuration:

MAC aging interval    : 300
MAC learning          : Enabled
MAC statistics        : Enabled
MAC limit Count       : 393215
MAC limit hit         : Disabled
MAC packet action drop: Disabled
LE  aging time        : 1200
LE  BD aging time     : 1200
```

Here you can see that the "MAC statistics" is Enabled and Junos is keeping track of each MAC address in the FIB. Before moving on, let's see how to enable a MAC account on a specific bridge domain:

```
bridge-domains {
    BD333_444 {
        vlan-tags outer 333 inner 444;
        interface ae1.1000;
        interface xe-2/1/1.300;
        interface ae0.333;
        bridge-options {
            mac-statistics;
        }
    }
}
```

All you need to do is enable the `bridge-options` for `mac-statistics` and you're good to go. Yet again, trust, but verify:

```
{master}
dhanks@R1-RE0> show l2-learning instance
Information for routing instance and bridge domain:

Flags (DL -disable learning, SE -stats enabled,
       AD -packet action drop, LH -mac limit hit)

Inst  Logical    Routing            Bridging      Index  IRB      Flags     BD
Type  System     Instance           Domain               Index              vlan
BD    Default    default-switch     BD333_444     8534            SE        333,444
```

And here you can see that the bridge domain `BD333_444` has the SE flag set. You can see the legend on the top indicated that `SE` means "stats enabled."

Now that the MAC account is turned on, what's the big deal? Let's take a look at a MAC address in the FIB:

```
{master}
dhanks@R1-RE0>
show route forwarding-table bridge-domain BD333_444 extensive destination
  5c:5e:ab:72:c0:82/48
Routing table: default-switch.bridge [Index 4]
Bridging domain: BD333_444.bridge [Index 8534]
VPLS:

Destination:  5c:5e:ab:72:c0:82/48
  Learn VLAN: 0                        Route type: user
  Route reference: 0                   Route interface-index: 343
  IFL generation: 861                  Epoch: 29
  Sequence Number: 14                  Learn Mask: 0x00000004
  L2 Flags: accounting
  Flags: sent to PFE
  Next-hop type: unicast               Index: 558      Reference: 6
  Next-hop interface: ae0.333
  Route used as destination:
    Packet count:                    0  Byte count:                     0
  Route used as source:
    Packet count:                   23  Byte count:                  1564
```

With MAC accounting turned on you get two additional fields: *route used as destination* and *route used as source*. Each field also keeps track of the number of packets and bytes. In this specific example, the book's lab used a CE device to `ping 255.255.255.255` out the interface going to the MX router R1. You can see that the MAC address of the CE is `5c:5e:ab:72:c0:82`, and it has been used as a source 23 times. Pinging the broadcast address `255.255.255.255` from the CE was just an easy way to guarantee that frames would be destined toward R1 without the chance of a reply. Otherwise, if you pinged a regular address successfully, the packet count for "route used as destination" and "route used as source" would be equal.

Integrated Routing and Bridging

How does one take the quantum leap from Layer 2 into Layer 3? The secret is that you need a gateway that has access to a Routing Information Base (RIB) and sits in the same bridge domain as you. The Junos way of making this happen is through a logical interface called `irb`. The astute reader knows that `irb` stands for *Integrated Routing and Bridging*.

Although IRB is a bit more than a simple gateway, it has other features such as handling control packets for routing protocols such as OSPF, IS-IS, and BGP. If you're running a multicast network, it will also handle the copying of frames for the bridge domain.

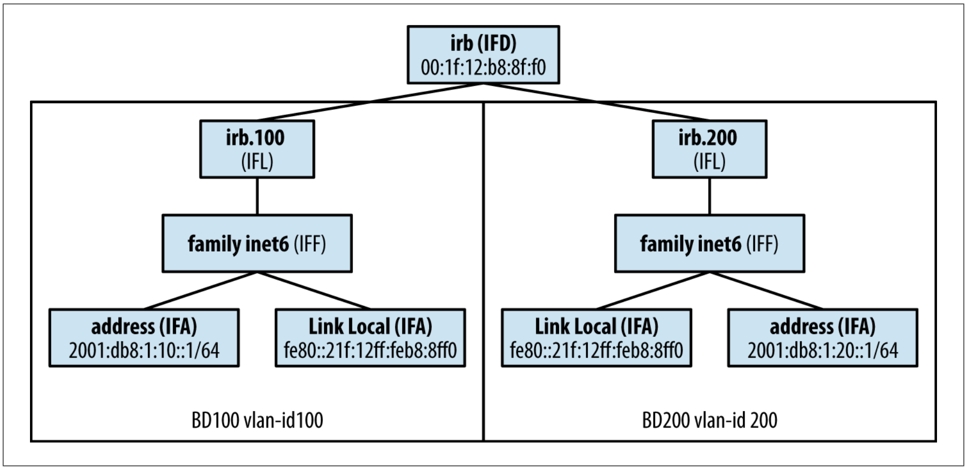


Figure 2-38. Illustration of IRB and Bridge-Domain Integration.

The hierarchy of Figure 2-38 should look familiar, as it follows the same interface hierarchy discussed previously in the chapter. At the top of the hierarchy sits the interface `irb`—this is a pseudo interface inside of Junos that acts as the gateway between bridge domains at the RIB. The `irb` is able to do this because it has both Layer 2 and Layer 3 that are associated to bridge domains and route tables. Let's take a look at a basic example:

```
interfaces {
        irb {
            unit 100 {
                family inet6 {
                    address 2001:db8:1:10::1/64;
                }
            }
            unit 200 {
                family inet6 {
                    address 2001:db8:1:20::1/64;
                }
            }
        }
    }
    bridge-domains {
        BD100 {
            vlan-id 100;
            routing-interface irb.100;
        }
        BD200 {
            vlan-id 200;
            routing-interface irb.200;
        }
    }
```

This example demonstrates how to configure the two bridge domains `BD100` and `BD200` to use the **`irb`** interface for routing. The magic is in the keyword **`routing-interface`**, specifying which `irb` IFL the bridge domain should use for routing.

For example, hosts that sit in `BD1000` and an IPv6 address in the `2001:db8:1:10::/64` range and use `2001:db8:1:10::1` as the gateway would be able to route outside of the bridge domain. A good example would be a host sitting in `BD100` and trying to ping the `irb.200` IFA `2001:db8:1:20::1` that sits in `BD200`.

IRB Attributes

The `irb` interface has a couple attributes that are automatically calculated: Maximum Transmission Unit (MTU) and interface speed. However, these values can be configured manually and override the defaults. The `irb` interface speed is set to 1G by default. There's no automatic calculation that happens, and it's purely cosmetic, as it doesn't place any sort of restrictions on the actual speed of the interface.

The `irb` IFL MTU is automatically calculated by Junos by looking at all of the IFLs in the bridge domain that specifies the **`routing-interface`** of the **`irb`**. For example, if `BD100` specified `irb.100` as the `routing-interface`, Junos will look at all of the IFLs associated with the bridge domain `BD100`. The `irb` IFL MTU will automatically be set to the lowest MTU of any of the IFLs of the corresponding bridge domain. Let's look at an example:

```
    interfaces {
        ae0 {
            vlan-tagging;
            mtu 8888;
            unit 0 {
                family bridge {
                    interface-mode trunk;
                    vlan-id-list 1-999;
                }
            }
        }
        ae1 {
            mtu 7777;
            unit 0 {
                family bridge {
                    interface-mode trunk;
                    vlan-id-list 1-999;
                }
            }
        }
        ae2 {
            mtu 6666;
            unit 0 {
                family bridge {
                    interface-mode trunk;
                    vlan-id-list 1-999;
                }
            }
        }
    }
    bridge-domains {
        BD100 {
            vlan-id 100;
            routing-interface irb.100;
        }
        BD200 {
            vlan-id 200;
            routing-interface irb.200;
        }
    }
```

In this example, you can see that interfaces `ae0`, `ae1` and `ae2` are using the Enterprise-style interface configuration and are members of both bridge domains `BD100` and `BD200`. You can also see that `BD100` is using `irb.100` as the `routing-interface`, and `BD200` is using `irb.200` as the `routing-interface`. The interface `ae2` has the lowest MTU of all three interfaces. Let's take a look at the `show interfaces irb` command; you should see that both `irb` IFLs will have a MTU of 6666.

```
{master}
dhanks@R1-RE0> show interfaces irb
Physical interface: irb, Enabled, Physical link is Up
  Interface index: 142, SNMP ifIndex: 1191
  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: 00:1f:12:b8:8f:f0, Hardware address: 00:1f:12:b8:8f:f0
```

```
    Last flapped   : Never
      Input packets : 0
      Output packets: 0

    Logical interface irb.100 (Index 330) (SNMP ifIndex 1136)
      Flags: SNMP-Traps 0x4004000 Encapsulation: ENET2
      Bandwidth: 1000mbps
      Routing Instance: default-switch Bridging Domain: VLAN100+100
      Input packets : 348
      Output packets: 66278
      Protocol inet, MTU: 6652
        Flags: Sendbcast-pkt-to-re
          Destination: 192.0.2.0/26, Local: 192.0.2.1, Broadcast: 192.0.2.63
        Addresses, Flags: Is-Preferred Is-Primary
          Destination: 192.0.2.0/26, Local: 192.0.2.2, Broadcast: 192.0.2.63
      Protocol multiservice, MTU: 6652

    Logical interface irb.200 (Index 349) (SNMP ifIndex 5557)
      Flags: SNMP-Traps 0x4004000 Encapsulation: ENET2
      Bandwidth: 1000mbps
      Routing Instance: default-switch Bridging Domain: VLAN200+200
      Input packets : 3389
      Output packets: 63567
      Protocol inet, MTU: 6652
        Flags: Sendbcast-pkt-to-re
        Addresses, Flags: Is-Preferred Is-Primary
          Destination: 192.0.2.64/26, Local: 192.0.2.66, Broadcast: 192.0.2.127
      Protocol multiservice, MTU: 6652
```

That's interesting. It would be logical to expect the MTU of `irb.100` and `irb.200` to be 6666, but as you can see it's clearly set to 6652. What happened is that Junos automatically subtracted the Layer 2 header, which is 14 bytes (MAC addresses + Ether-Type), from the lowest MTU of any IFLs in the bridge domain, which was 6666. This is how you come up with an MTU of 6652.

Virtual Switch

Now that you have all of the pieces of the MX puzzle, let's put them together and *virtualize* it. Recall from the very beginning of the chapter that the MX supports multiple Layer 2 networks, which is done via a feature called a *routing instance*. Each routing instance must have an instance type, and when it comes to virtualizing Layer 2, the instance type will be a *virtual switch*.

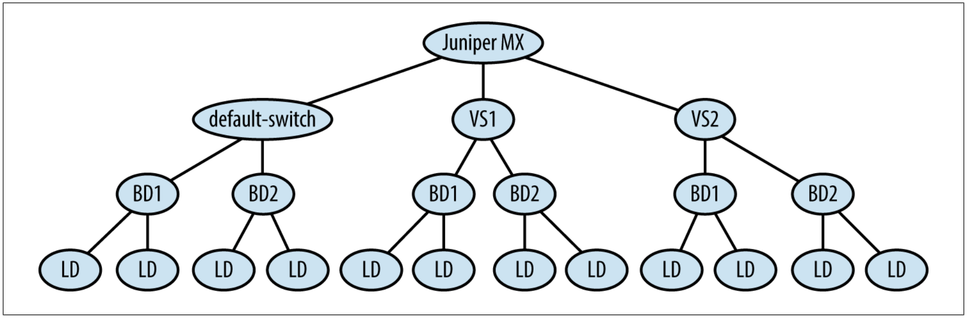


Figure 2-39. Virtual Switch Hierarchy.

In Figure 2-39, VS1 and VS2 represent routing instances with the type of virtual switch, while the default routing instance is referred to as the *default-switch routing instance*. Notice that each instance is able to have overlapping bridge domains because each routing instance has its own namespace. For example, bridge domain BD1 is present in the default-switch, VS1, and VS2 routing instances, but each bridge domain has its own learning domain per bridge domain.

Perhaps the real question is, what is the use case for virtual switches? This question goes back to the beginning of the chapter to help address scaling and isolation challenges. Perhaps your network provides Ethernet-based services to many different customers. Well, virtual switches are a great tool to segment customers and provide scale, qualified learning, and overlapping bridge domains. Each virtual switch is able to have its own set of IFLs, route tables, bridge domains, and learning domains.

Configuration

For those of you that are already familiar with routing instances, you'll find that creating virtual switches is very easy. Instead of showing you how to simply create a virtual switch, let's make it more interesting and actually migrate an existing Layer 2 network from the default-switch routing instance into its own private virtual switch. Let's take a look at what there is to start with:

```
interfaces {
    ae0 {
        vlan-tagging;
        unit 0 {
            family bridge {
                interface-mode trunk;
                vlan-id-list 1-999;
            }
        }
    }
    ae1 {
        unit 0 {
            family bridge {
                interface-mode trunk;
                vlan-id-list 1-999;
            }
        }
    }
    ae2 {
        unit 0 {
            family bridge {
                interface-mode trunk;
                vlan-id-list 1-999;
            }
        }
    }
}
```

There are three vanilla IEEE 802.1Q trunk interfaces accepting VLANs 1 through 999. Nothing fancy. You should even notice that these IFLs are configured using the Enterprise-style configuration. Let's take a look at any `irb` interfaces:

```
interfaces {
    irb {
        unit 100 {
            family inet {
                address 192.0.2.2/26;
            }
        }
        unit 200 {
            family inet {
                address 192.0.2.66/26;
            }
        }
    }
}
```

There are two `irb` IFLs—`irb.100` and `irb.200`—and each IFL has its own /26 IPv4 address. Let's see how these are set up in the bridge-domains:

```
bridge-domains {
    VLAN100 {
        vlan-id 100;
        routing-interface irb.100;
    }
    VLAN200 {
        vlan-id 200;
        routing-interface irb.200;
    }
}
```

These are two very basic bridge domains: `VLAN100` and `VLAN200`. Each bridge domain has its own `vlan-id` and references to the corresponding `irb` interface. Let's verify our findings with some show commands:

```
{master}
dhanks@R1-RE0> show bridge domain

Routing instance        Bridge domain            VLAN ID     Interfaces
default-switch          VLAN100                  100
```

```
                                                             ae0.0
                                                             ae1.0
                                                             ae2.0
default-switch          VLAN200                  200
                                                             ae0.0
                                                             ae1.0
                                                             ae2.0
```

Everything is as expected. The bridge domain, VLAN ID, and interfaces all match what you observed in the configuration. The most interesting field is the "Routing instance." Both `VLAN100` and `VLAN200` are part of the `default-switch` routing instance. Before you move on to the migration, you decide that it's best to check the forwarding table for `VLAN100`:

```
{master}
dhanks@R1-RE0> show route forwarding-table bridge-domain VLAN100
Routing table: default-switch.bridge
Bridging domain: VLAN100.bridge
VPLS:
Destination        Type RtRef Next hop           Type Index NhRef Netif
5c:5e:ab:6c:da:80/48 user     0                  ucst   561     7 ae1.0
5c:5e:ab:72:c0:80/48 user     0                  ucst   595     7 ae2.0
```

You confirm that the FIB indicates that the bridge domain `VLAN100` is part of the `default-bridge` routing instance. Content with this verification, you decide to move forward with the virtual switch migration.

What are the steps to migrate this basic bridging example from the `default-switch` routing instance into a new virtual switch?

1. Create a new `routing-instance` with the `instance-type` of `virtual-switch`.
2. Associate the IFLs `ae0.0`, `ae1.0`, and `ae2.0` with the new `routing-instance`.
3. Move the `bridge-domains` from the `default-switch` into the new `routing-instance`.

Let's go ahead and create the basic routing instance:

```
{master}[edit]
dhanks@R1-RE0# set routing-instances CUSTOMER-A instance-type virtual-switch
```

Now, add the IFLs `ae0.0`, `ae1.0`, and `ae2.0`:

```
routing-instances {
    CUSTOMER-A {
        instance-type virtual-switch;
        interface ae0.0;
        interface ae1.0;
        interface ae2.0;
    }
}
```

The tricky part is moving the existing bridge domains into the new virtual switch. Let's use the `load merge` command to make the job easier:

```
{master}[edit]
dhanks@R1-RE0# delete bridge-domains
```

```
{master}[edit]
dhanks@R1-RE0# edit routing-instances CUSTOMER-A

{master}[edit routing-instances CUSTOMER-A]
dhanks@R1-RE0# load merge terminal relative
[Type ^D at a new line to end input]
bridge-domains {
    VLAN100 {
        vlan-id 100;
        routing-interface irb.100;
    }
    VLAN200 {
        vlan-id 200;
        routing-interface irb.200;
    }
}
^D
load complete
```

Very cool; let's see what's there now:

```
routing-instances {

    CUSTOMER-A {
        instance-type virtual-switch;
        interface ae0.0;
        interface ae1.0;
        interface ae2.0;
        bridge-domains {

            VLAN100 {
                vlan-id 100;
                routing-interface irb.100;
            }

            VLAN200 {
                vlan-id 200;
                routing-interface irb.200;
            }
        }
    }
}
```

We've successfully created the new routing instance, associated the IFLs with the routing instance, and migrated the bridge domains into the routing instance. The astute reader might be curious about the `irb` interfaces. When creating virtual switches, there's no requirement to associate the `irb` interfaces with the routing instance. Junos takes care of this automatically for you.

Let's verify the behavior of this new virtual router with the same show commands as before.

```
{master}
dhanks@R1-RE0> show bridge domain

Routing instance        Bridge domain            VLAN ID     Interfaces
CUSTOMER-A              VLAN100                  100
                                                             ae0.0
                                                             ae1.0
                                                             ae2.0
CUSTOMER-A              VLAN200                  200
                                                             ae0.0
                                                             ae1.0
                                                             ae2.0
```

Very interesting; you now see that bridge domains VLAN100 and VLAN200 are part of the CUSTOMER-A routing instance instead of the default-switch. Let's review the FIB as well:

```
{master}
dhanks@R1-RE0> show route forwarding-table table CUSTOMER-A bridge-domain VLAN100
Routing table: CUSTOMER-A.bridge
Bridging domain: VLAN100.bridge
VPLS:
Destination        Type RtRef Next hop           Type Index NhRef Netif
5c:5e:ab:6c:da:80/48 user     0                  ucst   561     9 ae1.0
5c:5e:ab:72:c0:80/48 user     0                  ucst   557     9 ae2.0
```

Everything is working as expected. The FIB is also showing that bridge domain VLAN100 is using the routing table CUSTOMER-A.bridge.

Summary

If this is your first time with advanced bridging, take a look at the chapter review questions and see how well you do. If your results are mediocre, it may be beneficial to go back and reread this chapter again, as the advanced bridging features and concepts are tightly integrated. As you reread the chapter with this core knowledge behind your belt, you will have a new perspective and understanding and be able to grok advanced bridging.

The chapter started with the basics: Ethernet. Sometimes you take the basics for granted, don't review them for several years, and the details become fuzzy. It's important to fully understand Ethernet frame formats, including IEEE 802.1Q and IEEE 802.1QinQ, before moving into advanced bridging.

Next the chapter took a step back and pulled back the covers of the Junos interface hierarchy and introduced terms such as IFD, IFL, and IFF that are typically reserved for Juniper engineer employees. The Junos interface hierarchy is critical to fully understanding how advanced bridging on the MX works.

Giving a brief overview of the Enterprise Style versus Service Provider Style should give the reader a glimpse into the flexibility of the MX—it caters to all types of customers and networks. The Enterprise Style gives you the ability to write simple configurations to perform basic bridging, while the Service Provider Style takes bridging to another level and introduces VLAN mapping.

To firm up all your understanding of bridging, the chapter took a deep dive into bridge domains. You learned that there are many different types of bridge domains and how they're really just different ways to apply automatic VLAN mapping or solving interesting challenges. You should have also learned how to interact with bridge domains by enabling MAC limits, accounting, and other features.

The chapter then took the quantum leap from Layer 2 to Layer 3 with integrated routing and bridging. Such a simple, yet powerful feature; after all, what's the use of a bridge domain if you can never go outside of its boundaries and route?

To wrap things up and put your understanding into high gear, the chapter introduced a case study involving advanced bridging that didn't try to insult your intelligence with IPv4, but opted for IPv6 in each example.

Chapter Review Questions

1. How many VLAN tags are in an IEEE 802.1Q frame?
    a. 1
    b. 2
    c. 3
    d. All of the above
2. Which type of IFL VLAN tagging is required to support IEEE 802.1QinQ?
    a. `vlan-tagging`
    b. `stacked-vlan-tagging`
    c. `flexible-vlan-tagging`
    d. `vlan-tags`
3. When would you want to use `encapsulation flexible-ethernet-services`?
    a. Creating access ports
    b. To automatically set the encapsulation for each IFL to `vlan-bridge`
    c. To independently set the encapsulation for each IFL
    d. To support multiple customers from the same IFD
4. Does the Enterprise-style interface configuration support VLAN mapping?
    a. Yes
    b. No
5. If there was an ingress frame (C-TAG = 100) and you needed to perform VLAN mapping so that the egress frame was (S-TAG = 5, C-TAG = 444), which stack operation would you need for `input-vlan-map`?

    a. Swap-swap
    b. Pop-swap
    c. Swap-push
    d. Push-push

6. How many learning domains are in a bridge domain configured with vlan-id none?
    a. 0
    b. 1
    c. 4,000
    d. 4,094
7. How many learning domains are in a bridge domain configured with vlan-id all?
    a. 0
    b. 1
    c. 4,000
    d. 4,094
8. What does MAC accounting do?
    a. Counts the number of MAC addresses globally
    b. Counts the number of dropped ARP requests
    c. Keeps track of MAC address movement between IFLs in a bridge domain
    d. Keeps track of how many times a particular MAC address has been used for source or destination bridging
9. What's the default speed of the interface `irb`?
    a. 100 Mbps
    b. 1,000 Mbps
    c. Determined by the number of ports in the bridge domain
    d. Set to the highest port speed in the bridge domain
10. Can a bridge domain with the same vlan-id be configured across multiple virtual switches?
    a. Yes
    b. No

Chapter Review Answers

1. **Answer: A**. There's only a single VLAN ID in IEEE 802.1Q.
2. **Answer: B,C**. This is a tricky one, as both `stacked-vlan-tagging` and `flexible-vlan-tagging` will support IEEE 802.1QinQ frames.

3. **Answer: C,D**. When using `encapsulation flexible-ethernet-services`, this enables per IFL encapsulation, which is most often used when supporting multiple customers per port.
4. **Answer: A**. The Enterprise Style supports basic VLAN mapping via the `family bridge vlan-rewrite` function.
5. **Answer: C**. The ingress frame has a single C-TAG and the objective is to perform VLAN mapping to support both a S-TAG and a C-TAG. The tricky part of the requirement is that the original C-TAG is removed and the remaining S-TAG and a C-TAG are completely different VLAN IDs. The stack operation `swap-push` will swap the outer tag and push a new tag, resulting in the proper S-TAG and C-TAG.
6. **Answer: B**. When a bridge domain is configured with `vlan-id none`, it supports a single bridge domain and learning domain; it also requires that you use `input-vlan-map` and `output-vlan-map` to perform VLAN mapping.
7. **Answer: D**. When a bridge domain is configured with `vlan-id all`, it supports a single bridge domain with 4,094 learning domains.
8. **Answer: D**. MAC accounting will enable per MAC statistics that keep track of how many times it has been used as a source or destination address. Use the `show route forwarding-table` command to verify.
9. **Answer: B**. The interface `irb` is automatically set to a speed of 1,000 Mbps; however, this doesn't actually impact the performance of the interface because it's just a cosmetic attribute.
10. **Answer: A**. The answer is a definite *yes*. The entire purpose of a virtual switch is to provide a scalable method to create isolated Layer 2 networks.

CHAPTER 3

Stateless Filters, Hierarchical Policing, and Tri-Color Marking

This chapter covers stateless firewall filters and policers on MX routers. The MX Series has some special features and hardware that can make firewall filters and policers not only stronger, faster, and smarter, but also, once you get the hang of their operation, easier. So even if you think you know how to protect the routing engine, don't skip this chapter or the next. The MX Series is one awesome piece of iron, and users are always finding new ways to deploy its features for revenue. As critical infrastructure, it's well worth protecting; after all, the best rock stars have bodyguards these days.

By the way, this chapter is an overview, but is required reading for Chapter 4, where we blast right into case studies of IPv4 and IPv6 routing engine protection filters and coverage of the new DDoS policing feature available on Trio platforms. Chapter 4 is *not* going to pause to go back and reiterate the key concepts found here in Chapter 3.

The topics discussed in this chapter include:

- Firewall filtering and policing overview
- Filter operation
- Policer types and operation
- Filter and policer application points
- Transit filtering case study: Bridging with BUM protection

Firewall Filter and Policer Overview

The primary function of a firewall filter is to enhance security by blocking packets based on various match criteria. Filters are also used to perform multifield classification, a process whereby various fields in a packet (or frame) are inspected, with matching traffic being subjected to some specialized handling. For example, subjecting the traffic to a policer for rate limiting, assigning the traffic to a CoS forwarding class for later queuing and packet rewrite operations, or directing the traffic to a specific routing instance where it can be forwarded differently than nonmatching traffic to achieve what is known as Filter-Based Forwarding (FBF) in Junos, a concept akin to Policy-Based Routing (PBR) in other vendors' equipment.

Policers are used to meter and mark traffic in accordance to bandwidth and burst size settings. Policing at the edge enforces bandwidth-related SLAs and is a critical aspect of Differentiated Services (DS) when supporting real-time or high-priority traffic, as this is the primary mechanism to ensure that excess traffic cannot starve conforming traffic that is scheduled at a lower priority. In addition to discard actions, policers can mark (or color) out of conformance traffic, or alter its classification to place it into a new forwarding class.

Those familiar with the IOS way of doing things quickly recognize that stateless Junos filters provide functionality that is similar to Access Control Lists (ACLs), whereas policers provide rate enforcement through a mechanism that is similar to Committed Access Rate (CAR).

Stateless versus Stateful

Filters are categorized as being stateful or stateless based on whether they maintain connection or flow state tables versus simply treating each packet in a flow in a standalone manner. As with all things on Earth, there are advantages and disadvantages to both forms of filtering.

Stateless

As the name implies, a stateless filter does not maintain flow state or packet context beyond that of a single packet. There is no flow table, or, for that matter, no concept of a flow (with a flow being defined by some tuple such as Source Address, Destination Address, Protocol, and Ports). The upside is relatively low cost and raw performance, at near wire rate for all but the most complex of filter statements. All MX routers can perform stateless firewall functionality with no additional hardware or licenses needed.

The downside to a stateless filter is you have to either allow or deny a given packet, and the decision must be based solely on the information carried in the packet being processed. For example, if you expect to perform ping testing from your router, you will have to allow inbound ICMP Echo Reply packets in your filter. While you can place additional constraints on the allowed ICMP response, such as a specific source and destination addresses, whether fragmented packet replies are allowed, or the specific ICMP type, you still have to open a hole in the stateless filter for the expected reply traffic, and this hole remains open whether or not you have recently generated any requests.

Stateful

A stateful firewall (SFW) tracks session state and is capable of matching specific protocols requests to a corresponding reply. For example, it allows ICMP replies, but only when in response to a recently sent packet such as an ICMP echo request (ping). Rather than always allowing incoming ICMP echo replies, a SFW's flow table is dynamically created when an allowed outbound ICMP echo reply is detected, which begins a timer during which the response to that flow is permitted.

The added flow state allows for more sophisticated capabilities such as subjecting certain packets to additional scrutiny, a process known as Deep Packet Inspection, or by recognizing threats based on general anomaly detection or specific attack signatures, based on analyzing multiple packets in the context of a flow, something a stateless firewall can never do.

But, alas, nothing comes for free. An SFW is a high-touch device that requires a significant amount of RAM to house its flow tables, and processing power to plow through all those tables, which can easily become a performance bottleneck when taxed with too many flows, or simply too high of a data rate on any individual flow.

Trio-based MPCs can provide *in-line* services such as NAT and port mirroring without the need for additional hardware such as a services PIC. More demanding SFW and services-related function require the MX router be equipped with a MS-DPC to provide the hardware acceleration and flow state storage needed at scale.

SFW and related services like IPSec are beyond the scope of this book. However, the good news is that as the MS-DPC and Trio are Junos based, in-line services are configured and monitored using pretty much the same syntax and commands as used on the J-series Adaptive Service Module (ASM) or the M/T series Advanced Services PIC (ASP)/Multi-Services PICs, both of which are covered in *Junos Enterprise Routing*, Second Edition, also published by O'Reilly.

This chapter focuses on MX router support for stateless firewall filtering. Unless otherwise stated, all references to filters in this chapter are assumed to be in the context of a stateless filter.

Stateless Filter Components

Stateless filters can be broken down into five distinct components. These are filter types, protocol families, terms, match criteria, and the actions to be performed on matching traffic.

Stateless Filter Types

The Junos OS supports three different types of stateless filters: stateless, service filters, and simple filters. This chapter focuses on the stateless type because they are by far the most commonly deployed. While a detailed review is outside the scope of this book, a brief description of the other stateless filters types is provided for completeness.

Service Filter

A service filter is applied to logical interfaces that are configured on a services device such as a MX router's MS-DPC. The service filter is used to filter traffic prior to or after it has been processed by the related service set. Service filters are defined at the `[dynamic-profiles <profile-name> firewall family <family-name> service-filter]` hierarchy.

Simple Filter

Simple filters are available to provide some level of filtering support on FPCs that use commercial off-the-shelf (COTS) TCAM for firewall structures, namely IQ2 PICs and the MX's EQ-DPC, both of which are based on the EZChip. Simple filters are defined at the `[set firewall family inet simple-filter]` hierarchy. There are many restrictions to this type of filter because existing Junos match conditions were deemed too demanding for a TCAM-based engine; combined with their support on a limited and now long in the tooth hardware set, this explains why simple filters are rarely used. The restrictions for simple filters on MX routers are many:

- Simple filters are not supported on Modular Port Concentrator (MPC) interfaces, including Enhanced Queuing MPC interfaces.
- Simple filters are not supported for interfaces in an aggregated Ethernet bundle.
- You can apply simple filters to family `inet` traffic only. No other protocol family is supported.
- You can apply simple filters to ingress traffic only. Egress traffic is not supported.
- You can apply only a single simple filter to a supported logical interface. Input lists are not supported.
- On MX Series routers with the Enhanced Queuing DPC, simple filters do not support the `forwarding-class` match condition.
- Simple filters support only one source address and one destination address prefix for each filter term. If you configure multiple prefixes, only the last one is used.
- Simple filters do not support negated match conditions, such as the `protocol-except` match condition or the `except` keyword.
- Simple filters support a range of values for source and destination port match conditions only. For example, you can configure `source-port 400-500` or `destination-port 600-700`. With a conventional stateless filter you can match ports as a range, or list, such as `destination-port [ 20 73 90 ]`.
- Simple filters do not support noncontiguous mask values.

Protocol Families

You configure a filter under one of several protocol families to specify the type of traffic that is to be subjected to the filter. This action indirectly influences the possible match types, given that some match types are only possible for specific protocols and certain types of hardware. For example, a number of match conditions for VPLS traffic are supported only on the MX Series 3D Universal Edge Routers. Table 3-1 lists the protocol families supported by Trio filters:

Table 3-1. Supported Protocol Families for Filtering.

Nature of Traffic	Protocol Family	Comment
Protocol Agnostic	`family any`	All protocol families configured on a logical interface.
Internet Protocol version 4 (IPv4)	`family inet`	The family `inet` statement is optional for IPv4 as this is the default family.
Internet Protocol version 6 (IPv6)	`family inet6`	Use for IPv6 traffic.
MPLS/ MPLS-tagged IPv4	`family mpls`	Use when matching fields in one or more MPLS labels. For MPLS-tagged IPV4 traffic, family supports matching on IP addresses and ports in a stack of up to five MPLS labels with the `ip-version ipv4` keyword.
Virtual private LAN service (VPLS)	`family vpls`	Used to match VPLS traffic being tunneled over GRE or MPLS.
Layer 2 Circuit Cross-Connection	`family ccc`	Used for CCC style Layer 2 point-to-point connections.
Layer 2 Bridging	`family bridge`	Supported on MX Series routers only, used for bridged traffic.

Filter Terms

A firewall filter can contain one or many terms; at a minimum, at least one term must be present. Each term normally consists of two parts: a set of match criteria specified with a `from` keyword and a set of actions to be performed for matching traffic defined with a `then` keyword.

Consider the following filter named `tcp_out`:

```
[edit firewall filter tcp_out]
regress@halfpint# show

term 1 {
    from {
        protocol tcp;
        destination-port [ 23 21 20 ];
    }
    then {
        count tcp_out;
        accept;
    }
}
```

```
    term 2 {
        then count other;
    }
```

The tcp_out filter consists of two terms. The first term specifies a set of match conditions as part of the from statement. It's important to note that a logical AND is performed for each distinct match within a term, which is to say that all of the conditions shown on separate lines must be true for a match to be declared. In contrast, when you specify multiple matches of the same type, for example a sequence or range of destination port values, the grouping is shown on the same line in brackets and a logical OR function is performed.

In this case, the filter first tests for the TCP protocol by looking for a 06 in the IP packet's protocol field (which is the protocol number assigned to TCP by IANA), and then for either a 20, 21, or 23 in the destination port field. The nature of this type of match implies that we have also located an IP protocol packet at Layer 3. In fact, there is no protocol ip match option for an IPv4 filter; the IPv4 protocol is assumed by virtue of the filter being of the inet family, which is the default when no protocol family is defined within a firewall filter.

When a match is declared in the first term, the tcp_out counter is incremented and the traffic is accepted with no additional filter processing. All non-TCP traffic and all TCP traffic that does *not have* one of the specified destination ports will not match the first term and therefore falls through to the second term.

Note that the first term has both a from condition and a then action, while the second term only has a then action specified. The lack of a from condition is significant because it means we have a term with no match criteria, which means *all* traffic of the associated family will be deemed a match. In our example, this means that any IP-based traffic that is not matched in the first term will match the final term, and as a result increments the other counter.

> This seems like a good place for the standard warning about including a protocol match condition when you are also interested in matching on a protocol field such as a port. Had the operator not included the protocol tcp condition along with a port value in the telnet_out filter, it would be possible to match against UDP, or other non-TCP protocols such as ICMP, that just so happen to have the specified "port value" as part of their header or payload. Always take the time to specify as complete a set of match criteria as possible to avoid unpredictable filter behavior that is often very hard to fault isolate.

The Implicit Deny-All Term. It must be stressed that stateless filters always end with an implicit deny-all term that silently discards all traffic that reaches it. Some users may opt for a security philosophy in which you deny known bad traffic and then allow all else. In such a case, you must add an explicit accept-all term at the end of your stateless filter chain to ensure all remaining “good” traffic is accepted before it can hit the implicit deny-all.

Most users prefer a stronger security model where unknown threats are thwarted by specifically accepting known good traffic and denying all that remains. While this case can rely on the implicit deny-all term, most operators prefer to add their own explicit deny-all term, often with a counter or a log function added to assist in debugging and attack/anomaly recognition. The extra work involved for a final explicit term is trivial, but the added clarity can help avoid mistakes that often lead to outages of a valid service such as your routing or remote access protocols!

Filter Matching

There are a great many possible match criteria supported in Junos. You specify one or more match criteria within a term using the `from` keyword. Recall that a term with no `from` clause matches everything, and a term with multiple distinct match conditions results in an AND function, with a match only declared when all evaluate as true. The choice of filter family can influence what type of matches are available. For example, on a Trio-based MX router, the following match options are available for `family bridge` in release 11.4:

```
user@r1# set firewall family bridge filter foo term 1 from ?
Possible completions:
+ apply-groups         Groups from which to inherit configuration data
+ apply-groups-except  Don't inherit configuration data from these groups
> destination-mac-address  Destination MAC address
+ destination-port     Match TCP/UDP destination port
+ destination-port-except  Do not match TCP/UDP destination port
> destination-prefix-list  Match IP destination prefixes in named list
+ dscp                 Match Differentiated Services (DiffServ) code point
+ dscp-except          Do not match Differentiated Services (DiffServ) code point
+ ether-type           Match Ethernet type
+ ether-type-except    Do not match Ethernet type
+ forwarding-class     Match forwarding class
+ forwarding-class-except  Do not match forwarding class
+ icmp-code            Match ICMP message code
+ icmp-code-except     Do not match ICMP message code
+ icmp-type            Match ICMP message type
+ icmp-type-except     Do not match ICMP message type
> interface            Match interface name
+ interface-group      Match interface group
+ interface-group-except  Do not match interface group
> interface-set        Match interface in set
> ip-address           Match IP source or destination address
> ip-destination-address  Match IP destination address
+ ip-precedence        Match IP precedence value
+ ip-precedence-except  Do not match IP precedence value
+ ip-protocol          Match IP protocol type
+ ip-protocol-except   Do not match IP protocol type
> ip-source-address    Match IP source address
+ isid                 Match Internet Service ID
+ isid-dei             Match Internet Service ID DEI bit
```

```
    + isid-dei-except      Do not match Internet Service ID DEI bit
    + isid-except          Do not match Internet Service ID
    + isid-priority-code-point  Match Internet Service ID Priority Code Point
    + isid-priority-code-point-except  Do not match Internet Service ID Priority Code Point
    + learn-vlan-1p-priority  Match Learned 802.1p VLAN Priority
    + learn-vlan-1p-priority-except  Do not match Learned 802.1p VLAN Priority
    + learn-vlan-dei       Match User VLAN ID DEI bit
    + learn-vlan-dei-except  Do not match User VLAN ID DEI bit
    + learn-vlan-id        Match Learnt VLAN ID
    + learn-vlan-id-except  Do not match Learnt VLAN ID
    + loss-priority        Match Loss Priority
    + loss-priority-except  Do not match Loss Priority
    + port                 Match TCP/UDP source or destination port
    + port-except          Do not match TCP/UDP source or destination port
    > prefix-list          Match IP source or destination prefixes in named list
    > source-mac-address   Source MAC address
    + source-port          Match TCP/UDP source port
    + source-port-except   Do not match TCP/UDP source port
    > source-prefix-list   Match IP source prefixes in named list
      tcp-flags            Match TCP flags
    + traffic-type         Match Match traffic type
    + traffic-type-except  Do not match Match traffic type
    + user-vlan-1p-priority  Match User 802.1p VLAN Priority
    + user-vlan-1p-priority-except  Do not match User 802.1p VLAN Priority
    + user-vlan-id         Match User VLAN ID
    + user-vlan-id-except  Do not match User VLAN ID
    + vlan-ether-type      Match VLAN Ethernet type
    + vlan-ether-type-except  Do not match VLAN Ethernet type
```

Certainly a lengthy list of match options, and given that family bridge is a Layer 2 protocol family, one cannot help but be struck by the rich set of protocol match options at Layer 3 (IP) and Layer 4 (TCP/UDP). Trio's ability to peer deep (up to 256 bytes) into Layer 3 traffic, even when functioning as a bridge, really drives home the power and flexibility of the Trio PFE chipset!

This chapter, as well as others that build on filter or policer functionality, expose the reader to a variety of match conditions for common protocols families. Given there are so many options, it just does not make sense to go into all of them here; besides, all are documented in various user manuals. It bears mentioning that for the bridge family you specify a `user-vlan` match type when matching on (C) tags associated with access ports and the `learn-vlan` match type for trunk ports when matching on Service Provider (S) tags. For example, this link takes you to documentation that defines stateless filter match capabilities for each protocol family, as of Junos release v11.4:

> *http://www.juniper.net/techpubs/en_US/junos11.4/topics/concept/firewall-filter-stateless-guidelines-for-configuring.html#jd0e392*

A Word on Bit Field Matching. Many protocol field patterns can be matched using a predefined symbolic name that's designed to be user-friendly. For example, matching on the SYN bit in a TCP header can be done with the keyword `syn`, or a hexadecimal value 0x2. While it may impress your geeky friends to have a hex-based packet filter, it's considered best practice to use the symbolic names when they're available to help improve filter legibility and lessen the chances of a mistake. Table 3-2 shows some of the more common bit field match types and their keyword aliases.

Table 3-2. Text Synonyms.

Text Synonym	Match Equivalent	Common Use
first-fragment	Offset = 0, MF = 1	Match on the first fragment of a packet for counting and logging.
is-fragment	Offset does not equal zero	Protect from fragmented DOS attacks
tcp-established	ack or rst	Allow only established TCP sessions over an interface. This option does not implicitly check that the protocol is TCP. Combine with a protocol TCP match condition. Known as `established` in IOS ACLs.
tcp-initial	syn and not ack	Allow TCP sessions to be initiated in the outbound direction only. Combine with a protocol TCP match condition. Equal to TCP flags of `match-all +syn -ack` in IOS ACLs

Junos also supports a range of boolean operators such as negations and logical AND/OR functions. Consult the documentation for a complete list of available bit field matches and supported operations. This information is documented for the v11.4 release at: *http://www.juniper.net/techpubs/en_US/junos11.4/topics/concept/firewall-filter-stateless-match-conditions-bit-field-values.html*.

Filter Actions

You use a `then` statement to list one or more actions that are to be performed on traffic that matches a filter term. If you omit the `then` keyword, the default action is to accept the matching traffic. Besides the basic `discard` and `accept` actions, filter can also mark packets to influence CoS processing, alter the traffic's drop probability, or count/log matches to assist in anomaly detection or to aid in troubleshooting filter operation. With Junos, you have the flexibility of specifying multiple actions in a single term, or you can use the `next-term` flow-control action (detailed in a later section) to have the same traffic match against multiple terms, where each such term has its own specific action.

Filters versus Routing Policy

Readers familiar with Junos routing policy will recognize many similarities between filter and policy configuration syntax and processing logic. Both filters and policy statements can consist of one or more terms (though a policy is allowed to have a single unnamed term and filters require all terms be explicitly named), with those terms based on a set of match criteria along with an associated action. They both support the concept of chaining, where you link small pieces of modular code to act as if it were a single, large piece filter or policy. Both support application for inbound or outbound flows, in the data and control planes, respectively, and both support the concept of a "default action" for traffic that's not been explicitly matched by a preceding term.

While so very similar, routing policy and filters perform two completely different functions that can sometimes achieve similar effects, and so their differences warrant a brief discussion here. Filters operate in the data plane, whereas policy functions in the Control Plane (CP). Filters can affect transit traffic *directly*, regardless of whether there is a viable next-hop for the traffic, by filtering the traffic itself. In contrast, policy can affect transit traffic *indirectly* by virtue of allowing a given next-hop, or not, in the routing table through filtering of route updates. Filters can affect traffic to the router itself, or transit traffic destined for a remote machine. Policy always acts on traffic destined to the local host in the form of a routing protocol update. Note that filtering such a route update at ingress can in turn impact how traffic in the data plane egresses the local router, but this is indirect compared to a filter's direct action in the data plane.

With regards to traffic destined to the local host, filters are used primarily for security whereas policy is used to influence the routing table and to control what routes you advertise to downstream peers. A comparison of the two features is given in Table 3-3.

Table 3-3. Firewall Filters versus Routing Policies.

Feature	Firewall filter	Routing policy
Operates in . . .	Forwarding plane	Control plane
Match keyword	from	from
Action keyword	then	then
Match attributes	Packet fields	Routes and their attributes
Default action	Discard	Depends on default policy of each particular routing protocol
Applied to . . .	Interfaces/Forwarding Tables	Routing protocols/tables
Named terms required	Yes	No
Chains allowed	Yes	Yes
Absence of from	Match all	Match all
Boolean operations when applied	No	Yes

As a final comparison point, consider the goal of wanting to prevent a given BGP update from entering your router. You could use a filter to block all BGP, or perhaps just BGP from a given peer. But this is a heavy hammer, as it affects all the routes that peer can ever advertise. Policy, on the other hand, operates on the individual routes that are received via routing protocol update, which in turn allows per-prefix control for filtering or attribute modification.

Like peanut butter and chocolate, the two go great together. You deploy filters to block unsupported routing protocols while also using policy to filter individual route updates from within those supported protocols.

Filter Scaling

Everything has a limit, and pushing anything too far will affect its performance, sometimes dramatically. Customers often ask questions like “How many terms can I have in a filter?” or “How many filters can I apply at any one time?” While reasonable, this type of question is hard to answer because filtering is only one dimension of system scaling, and most production networks deploy nodes that must scale in multiple dimensions simultaneously. As is so often true, you cannot have your cake and also keep it in safe storage for later consumption, which in this case means if you are pushing the box to scale with large numbers of BGP peers and millions of route prefixes, then there is a good chance you will hit some issue with filter scaling before the theoretical limit. It’s always a good idea to monitor system load, and to be on the lookout for error messages in the logs that warn of impending resource exhaustion, when pushing any router to high scales.

Each Trio PFE has a large pool of External Data Memory (EDMEM) for storing next-hop FIB entries, filter structures, Layer 2 rewrite data, and so on. While portions of this memory are reserved for each function, the memory system is flexible and allows areas with heavy use to expand into unused memory. As a result, it’s not uncommon to check some resource usage and find it seems alarming high, say 98 percent utilized, only to find you can keep pushing that scale dimension and later find the pool has been resized dynamically.

The command shown in the following is used to display the current allocations of EDMEM on a Trio MPC, which in this example is demonstrated on a system that is scaled to 3,000 EBGP/OSPF/RIP VRF peers, with each of the 2,000 VRF IFLs using an input filter for COS classification and policing.

Here, we look at the first PFE on FPC 5 (MPC 5). Note how the `request pfe execute` command is used to avoid having to drop to a shell and/or VTY to the desired PFE component. pretty cool:

```
{master}
regress@halfpint> request pfe execute target fpc5 command "show jnh 0 pool usage"
SENT: Ukern command: show jnh 0 pool usage
GOT:
GOT: EDMEM overall usage:
GOT: [NH///////////////|FW///|CNTR////////|HASH/////|ENCAPS////|--------------]
GOT: 0                 7.0  9.0          14.0      21.8       25.9            32.0M
GOT:
GOT: Next Hop
GOT: [**************************************************|--] 7.0M (98% | 2%)
GOT:
GOT: Firewall
GOT: [|--------------------] 2.0M (1% | 99%)
GOT:
GOT: Counters
GOT: [|----------------------------------------] 5.0M (1% | 99%)
GOT:
GOT: HASH
```

```
GOT: [*************************************************************] 7.8M (100% | 0%)
GOT:
GOT: ENCAPS
GOT: [*****************************************] 4.1M (100% | 0%)
GOT:
LOCAL: End of file
```

This display is based on the allocation of Mega Double Words (MDW), with each word being 32 bits/4 bytes in length--thus a DMW is 64 bits. In this display, 1 M equates to 1 MDW or 1 M * 8 B, which equals 8 MB (or 64 Mb). Here we can see that the portion of EDMEM allocated to filter structures is relatively small, at 2 MDW, when compared to the 7 MDW allocated for next hops. As noted previously, this setup has 2K (interface-specific) filters and policers in effect, along with a heavy BGP/VRF/route load, and clearly there is still room to grow with additional filter or policer structures. The current Trio PFE can allocate up to 88 MB per PFE to hold filter and policer structures.

You can display overall PFE memory usage with the summary option:

```
{master}
regress@halfpint> request pfe execute target fpc5 command "show jnh 0 pool summary"
SENT: Ukern command: show jnh 0 pool summary
GOT:
GOT:                 Name        Size      Allocated     % Utilization
GOT:                EDMEM    33554432       19734369               58%
GOT:                IDMEM      323584         201465               62%
GOT:                 OMEM    33554432       33079304               98%
GOT:          Shared LMEM         512             64               12%
LOCAL: End of file
```

The current scaling limits for Trio-based MPCs are shown in Table 3-4. In some cases, scale testing is still being conducted so preliminary numbers are listed in the interim. It's expected that Trio-based PFEs will outperform I-Chip systems in all dimensions.

Table 3-4. MX Trio versus I-Chip Filter Scale.

Parameters	Per Trio/MPC/Chassis	Per I-Chip 3.0/DPC/Chassis
Maximum number of interface policers	39 K (per chassis)	39 K (per chassis)
Maximum number of interface filters	16 K	16 K
Maximum 1-tuple terms	256 K	64 K
Maximum 1-tuple terms in a single filter	256 K	250 K

Trio platforms running Junos v11.4 can support up to 128 K policers per chassis when the policers are called from a firewall filter, as opposed to being directly applied to the IFL, where the current limit is up to 39 K policers per chassis.

Filter Optimization Tips

There is no free lunch, and this maxim holds true even for Junos and the Trio chipset. Forwarding performance can be impacted when high-speed interfaces have the majority of their packets evaluated by the majority of terms in a large filter, something that can occur when a filter is not laid out in an optimal fashion. Each term evaluated has a computation cost and a corresponding delay, so ideally a filter will be constructed such that the majority of packets meet a terminating action (accept or discard) early in the term processing. This is one reason why extensive use of the `next term` action modifier can impact filter throughput, because it forces the same packets to be evaluated by at least two terms.

Junos cannot optimize the ordering of terms in a filter, as this is a function of the traffic that is allowed versus disallowed, and a function of the specific traffic encounter in each network. As an extreme example, consider a case where ICMP echo traffic is to be accepted. Having the accept ICMP term at the end of a 1,000 term filter, in a network that experiences a lot of ping traffic, will likely show a performance impact given that each ICMP packet must be evaluated by the preceding 999 terms before it's finally accepted. In this case, moving the accept ICMP term to the beginning of the filter will dramatically improve performance, especially when under a heavy ICMP traffic load.

In addition to `next term`, heavy use of noncontiguous match conditions should be minimized, as such expressions are sometimes compiled into multiple terms for actual filter execution, which can negatively impact performance because a single term with a noncontiguous match effectively results in a longer filter with more terms to evaluate. Whenever possible, it's always best to use a more specific match value or to specify a contiguous set of matching criteria. For example, this is a poorly written port range match criteria that is likely to result in multiple terms given the non-contiguous range of numeric match conditions:

```
term 1 {
    from {
        port [ 1-10 20-60 13 ];
    }
}
```

Where possible, consider separate terms for each noncontiguous value, or better yet, specify a contiguous set of matching values:

```
term 1 {
    from {
        port [ 1-60 ];
    }
}
```

Avoiding heavy use of `next term` and discontiguous numeric range match criteria, and designing filter terms so that most traffic is accepted or discarded early in the process, are ways to help ensure that forwarding performance remaining near wire-rate, even when complex filter are in use.

Filtering Differences for MPC versus DPC

Readers familiar with the older ADPC-style line cards should keep the differences shown in Table 3-5 in mind when deploying filters on Trio based MX routers.

Table 3-5. MPC versus DPC Filter Processing.

Trio MPC	I-chip DPC
Egress filters apply to all traffic on Trio—L3 multicast and L2 BUM included.	Egress filters apply to L3 known Unicast only
Unknown unicast as an input interface filter match condition is not supported. On Trio, a MAC lookup is required to determine if a packet is an unknown unicast packet. Therefore, to support unknown unicast match, a BUM filter must be configured in the VPLS or bridge instance as applicable.	Unknown Unicast is supported
Simple filters are not supported on Trio.	Simple filters supported
Egress filters will use the protocol of the packet after lookup, not the incoming interface protocol.	Egress filters use ingress interface protocol
Can reset/set DSCP at ingress using filter with `dscp [0 | value]` action modifier.	DSCP can be reset/set using CoS rewrite on egress interface

Most of the differences are straightforward. The point about egress filters acting on the egress versus ingress protocol type for Trio versus I-chip is significant and warrants additional clarification. Imagine an MPLS transport-based VPLS scenario where the egress PE is a Trio-based MX router. The VPLS traffic received from remote PEs ingress from the core as type MPLS, and after MAC lookup in the LU chip, egress to the attached CE as protocol `vpls`; in an L3VPN, the traffic egresses as type `inet` or `inet6`.

If the goal is to apply a filter to PE-CE egress traffic, then on a Trio-based MX you will apply a filter of the `vpls` family. In contrast, on an I-Chip PFE, an egress filter using the `vpls` family (or `inet`/`inet6` in the case of a L3VPN) *does not apply* as the traffic is still seen as belonging to the `mpls` family. Because of this, you need to use a `vt` (tunnel) interface or `vrf-table-label` on the egress PE, both of which result in the packet being looped back around for a second lookup, to facilitate IP-level filtering at the egress of a VPN.

Enhanced Filter Mode

Enhanced filters are a Trio-only feature that is dependent on the chassis running in enhanced network services mode, which in v11.4 means you must include the `enhanced-ip` statement at the `[edit chassis network-services]` hierarchy. Enhanced filters are designed to save memory when an MX router is used for Subscriber access where you may need to support filtering for up to 250,000 customers.

Setting `enhanced-ip` mode will result in the powering off of any DPCs that are installed in the system. Only Trio-based MPCs can come online in this mode in v11.4.

Normally, a stateless filter is generated in both term-based mode for use on the ASICs and in a compiled format used by the kernel. The compiled version of the filter evaluates traffic flowing to or from the RE via the kernel. Enhanced mode filters save kernel memory as they are only used in the term-based ASIC format with no copies compiled for use in the kernel. For more information on enhanced mode filters in v11.4, refer to the following link: *http://www.juniper.net/techpubs/en_US/junos11.4/topics/reference/configuration-statement/enhanced-mode-edit-firewall.html*.

Note that any filter applied to the loopback or management interfaces has to be compiled into both formats, which means the enhanced setting is ignored for such filters. If you designate a filter as enhanced a commit warning is generated if the requisite chassis mode is not set:

```
[edit]
jnpr@R1-RE0# set firewall family inet filter accept-web enhanced-mode

[edit]
jnpr@R1-RE0# commit
re0:
[edit firewall family inet]
  'filter accept-web'
    warning: enhanced-mode defined for filter (accept-web) is inconsistent with the
             running configuration of the chassis network services (NORMAL (All FPC))
configuration check succeeds
re1:
commit complete
re0:
commit complete
```

Filter Operation

This section takes a deep dive into the operation and capabilities of firewall filters on MX routers. Ready, set, go.

Stateless Filter Processing

A firewall filter consists of one or more terms, with each term typically having both a set of match criteria and a set of actions to be performed on matching traffic. Traffic is evaluated against each term in the order listed until a match is found with a terminating action. Figure 3-1 illustrates these filter processing rules.

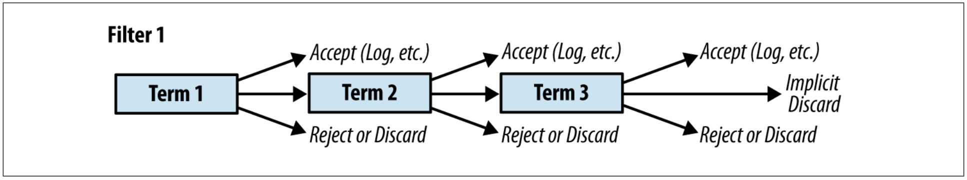


Figure 3-1. Filter Processing.

The traffic being evaluated begins at term 1, on the left, and makes its way toward the right through each successive term until a match is found, at which point the associated actions are carried out. Terminating actions are shown on the bottom of each filter term while nonterminating (action modifiers) are shown at the top. As was noted previously, traffic that does not match any of the configured terms is subjected to an implicit deny-all term that, as it name might imply, matches on all remaining traffic and directs it to a discard action.

While the block diagram is useful, there is nothing like dealing with actual filter syntax to help drive these processing points home. Consider the multiterm firewall filter called `EF_limit_G=768K`:

```
filter EF_limit_G=768K {
    term EF {
        from {
            forwarding-class EF;
        }
        then policer POL_EF_G=768K;
    }
    term default {
        then accept;
    }
}
```

Here, the first term has been called `EF`, in keeping with its function to match on traffic that has been classified into the Expedited Forwarding (EF) class. Because all EF traffic matches this term, it is subjected to a policer named `POL_EF_G=768K`. While not shown, you can assume that the policer is set to discard traffic that is in excess of its configured bandwidth settings; the policer discard action is most certainly terminating for the traffic that is deemed to be in excess. In contrast, in-profile traffic is handed back to the `EF` term, where it's *implicitly* accepted in the example given. The concept of implicit termination is important and is described in detail in the following.

Given that only EF traffic can match the first term, all non-EF traffic falls on through to the next term, which in this example is called `default.` This term's name is selected in accordance with its function as an explicit catch-all term for any traffic not previously matched and subjected to a termination action. Because this term has no match criteria specified, it matches on all traffic and thereby avoids any possibility of traffic inadvertently falling through to the implicit deny-all function present at the end of all Junos filters.

Filter Actions

When a packet matches a filter term, the associated action can be classified as terminating, nonterminating, or as flow-control. As the name implies, terminating actions cease filter processing at the point they are encountered with any remaining filter terms ignored and unprocessed. This means you must be extremely careful in the way that you order your filters and their terms; a given packet can only meet one terminating action, and it's going to be the action specified by the first term the packet matches. As an extreme example, consider the case of a 1,000-term filter named `foo` that begins with the following term:

```
filter foo {
    term explicit-deny {
        then {
            discard;
        }
    }
    term allow_ssh {
        from {
            protocol tcp;
        . . . .
```

Given that the first term to be evaluated matches all traffic with a terminating condition, it should be obvious why none of the other 999 terms are ever evaluated, and why you have so many angry customers calling you . . .

> This is a classic use case for the `insert` feature of the Junos CLI. Rather than having to delete and redefine an entire filter (or policy) when you need to reorder terms, you can simply tell Junos to insert an existing term before or after any other term in the filter (or policy).

Terminating Actions.

`accept`
: Accepts the packet for route lookup and subsequent forwarding to the selected next hop, which can be an external destination or the local host itself depending on where the filter is applied.

`discard`
: Silently discards the packet, which is to say that no ICMP error messages are generated to the source of the offending packet. This is the preferred option when you expect to discard traffic at scale, given that no PFE resources are taxed during a silent discard.

`reject`
: Discard the packet and generate an ICMP error message back to its source; while the specific type of ICMP error message sent depends on the specific configuration, the default message type is administratively prohibited. These error messages are generated within the PFE itself, and so do not consume RE compute cycles or any

of the bandwidth reserved for PFE to RE communications. In addition, the error messages are rate limited by the Junos microkernel (within the PFE), to guard against excessive resource consumption, but even so it's generally best practice to use the discard action to completely eliminate a potential DDoS vector that can be triggered remotely by flooding a network with large volumes of bogus traffic. It should be noted that source address spoofing is the norm in this kind of attack, such that any error messages you do bother to generate actually flow to the innocent (and legitimate) owner of the spoofed address blocks. If such error messages are being generated in large volumes, the error messages themselves can serve as a DDoS vector, but now with the nodes *in your network* seen as the attacker!

Nonterminating Actions. Nonterminating actions (also known as action modifiers) are functions that by themselves do not terminate filter processing, but instead evoke some additional action such as incrementing a counter or creating a syslog entry. Action modifiers are pretty straightforward with one exception; just like a term that has no `then` action, specifying only action modifiers in a term results in an implicit accept action. When you wish to use an action modifier without also accepting the matching traffic, you must specify an explicit terminating action of discard or reject, or use a flow control action to allow the packet to be processed by subsequent terms.

`count`

Increments the named counter for each matching packet of the specified protocol family. You view the counter results with a CLI `show firewall filter` command. Counters can consume PFE resources, so it's best not to use them "just because," especially so if used in a catch-all term that accepts large volumes of permitted traffic, where a high packet count does not tell you a lot anyway. In a production network, it's best to use counters for discard terms that are expected to have a relatively low volume of hits as an aid in troubleshooting any connectivity issues that may arise from packet discard. Counters are often added during initial filter deployment to help confirm expected filter matching and facilitate troubleshooting when unexpected results are found. Such counters are typically deactivated or removed from the configuration when the filter is confirmed to work properly and is placed into production.

`log`

Log the packet header information in a ring buffer within the PFE. You can view the log hits by issuing the CLI `show firewall log` command. This action modifier is supported for the `inet` and `inet6` families only. Note that the log cache cannot be cleared and can hold about 400 entries before it wraps around to begin overwriting older entries. The log's contents are retained in the PFE until requested by the cli, so this modifier does not consume much in the way of RE bandwidth or compute cycles.

`syslog`

This modifier is similar to the previous one, except now each hit is sent from the PFE to the RE, where it is kept in the local (or remote) syslog for future analysis.

In order for filter hits to be logged, you must specify a local or remote syslog file, and you must include the firewall facility for that log. The local syslog example shown places all filter hits with info level or above into the `/var/log/filter_hits` log file. You can also write filter hits to the main syslog messages file by including the firewall facility, but placing this information into individual logs tends make analysis easier. This configuration will also result in the creation of 4 `filter_hits` log files, each with a size of 10m before filling up and overwriting the oldest entries:

```
file filter_hits {
    firewall info;
    archive size 10m files 4;
}
```

`policer`

Use this modifier to direct matching traffic to a policer for rate limiting. Junos policers are covered in detail later, but for now it's worth noting that a Junos policer automatically creates a counter to tally its packet discards. You can view the policer counter along with any other filter counters with a CLI `show firewall filter <name>` command.

`dscp`

This modifier allows you to rewrite DSCP bits in matching `inet` (IPv4) packets.

Support for this modifier is hardware and platform dependant, so it's best to check that your FPC/MPCs are supported, else expect silent fail.

`forwarding-class`

This modifier is used to set matching traffic's forwarding class (FC), which in turn is used to bind the packet to an egress queue and scheduler for CoS processing. Note that using a filter to set a FC is considered Multi-Field (MF) classification, given that the filer can match on various fields besides those strictly designated for ToS markings, and that an ingress MF classifier overwrites the FC that may have been set by a Behavior Aggregate (BA) classifier.

On Trio PFEs, you can use an egress filter to alter a packet's FC, thereby affecting its egress queuing and rewrite operations.

`loss-priority`

This modifier can set a packet's loss priority, which is used later in the face of congestion to make WRED-related discard decisions. You need to include the `tri-`

`color` statement at the `[edit class-of-service]` hierarchy if you want to set more than two levels of loss priority, otherwise only high and low are available.

`next-hop-group`
: This modifier is used to associate matching traffic with an interface group and is supported for the `inet` family only. Groups are used on the MX for port mirroring, where matching traffic can be sent out one or more of the interfaces bound to the specified group.

`port-mirror`
: This modifier tags matching traffic of the specified family for port mirroring. You must also configure the port mirroring interfaces for this to work.

`prefix-action`
: This modifier evokes the per-prefix policing and counting feature that's designed to make deploying large numbers of policers and counters, on a per-prefix level, as might be used to limit a college dorm's Internet activities, easy to deploy.

`sample`
: This modifier marks matching traffic as eligible for sampling to support flow collection and analysis using Jflow; Jflow is the Junos feature that exports cflowd formatted records (which is the same functionality as the widely known Netflow application and its v5 record export on other vendor's equipment). For this to work, you also need to configure sampling, which includes the statistical probability that such a marked packet will actually be sampled; 1:1 sampling is rarely done in production networks as port mirroring is better suited to sampling all traffic. Currently, sampling is supported for the `inet`, `inet6`, and `mpls` families only.

`three-color-policer`
: Similar to the policer modifier, except used to direct traffic to a single-rate two color (srTCM) or a two-rate three color policer (trTCM).

`traffic-class`
: This modifier allows you to rewrite the Traffic Class bits in matching `inet6` (IPv6) packets, similar to the `dcsp` modifier's functionality for IPv4.

> Support for this modifier is hardware and platform dependant, so it's best to check that your FPC/MPCs are supported, else expect silent fail.

Flow Control Actions. Flow control actions, as their name would imply, are used to alter the normal flow of filter processing logic.

`next-term`
: This modifier causes matching traffic to immediately pass through to the next term in the current filter. At first glance, this function may seem redundant, but it's not. Use this modifier to avoid any implicit-accept functions associated with use of an action-modifier, while still allowing the packet to be processed by additional filter

terms (when present). For example, assume you wish to match some traffic in a given term and evoke a count function. By default, matching traffic will be implicitly accepted unless you specify an explicit action. Here, the `next-term` action prevents implicit acceptance by the counting term, thus allowing the same traffic to be processed by additional terms.

> Heavy use of next-term should be avoided as it has a computational load associated with it and can affect performance with high-scaled firewall configurations.

Policing

As mentioned in the chapter overview, a stateless filter can evoke a traffic policer action to rate limit traffic according to user-specified bandwidth and burst size settings. Rate limiting is a critical component of a CoS architecture and a primary mechanism for ensuring that a broad range of Service Level Agreements (SLAs) can be honored in a modern, multiservice internetwork. In such networks, it's critical that you meter and limit ingress traffic flows to protect the shared resources in the network's core to ensure that each subscriber does not consume excessive bandwidth, leading to poor performance for users that honor their contracts.

Rate Limiting: Shaping or Policing?

The basic idea of rate limiting is rather straightforward. The goal is to limit the amount of traffic that can be sent by a given device in a given unit of time. Simple enough from 20,000 feet, but there are several ways to achieve rate limiting, namely shaping versus policing. While both provide similar effects at the macro level, they have distinct operational differences that can be seen at the micro level when looking at the way packets are placed onto the wire. Most pronounced is shaping introduced delay in an attempt to control loss, making them better suited for use with TCP-based applications, while a policer does the opposite, trading loss for delay, making it better suited to real-time applications.

Shaping

Traffic shaping attempts to reduce the potential for network congestion by smoothing out packet flows and regulating the rate and volume of traffic admitted to the network. In effect, a shaper endeavors to turn valleys and hills into a level road by buffering the peaks and using that excess traffic to fill up the valleys. There are two primary ways to facilitate traffic shaping:

The Leaky Bucket Algorithm. The leaky bucket algorithm provides traffic smoothing by presenting a steady stream of traffic to the network. Its operation is shown in Figure 3-2.

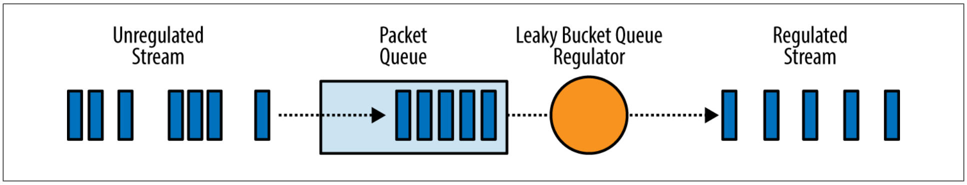


Figure 3-2. Leaky Bucket Shaping Algorithm.

The figure shows how the leaky bucket shaper turns a bursty stream of packets into a consistent stream consisting of equally spaced packets. In operation, the unregulated stream of ingress packets are placed into a queue that's controlled by a queue regulator. Bursts are controlled through immediate discard when the flow presents more packets than the queue can store. Packets at the head of the queue are forwarded at a constant rate determined by the configuration of the queue regulator's "drain rate." When properly configured, the packets should not be forwarded into the network at a rate greater than the network can, or is willing to, absorb. The length (or depth) of the packet queue bounds the amount of delay that a packet can incur at this traffic shaper. However, the end-to-end delay can be much longer if the packet should incur additional shapers, or develops queuing delays at downstream hops due to network congestion, given that shaping is typically performed only at ingress to the network.

The Token Bucket Algorithm. The token bucket algorithm is a type of long-term average traffic rate shaping tool that permits bursts of a predetermined size, with its output being a burst-regulated stream of traffic that is presented to the network. The token bucket rate limiting algorithm enforces a long-term average transmission rate while permitting bounded bursts. With this style of shaper, a token bucket is used to manage the queue regulator, which in turn controls the rate of packet flow into the network, as shown in Figure 3-3.

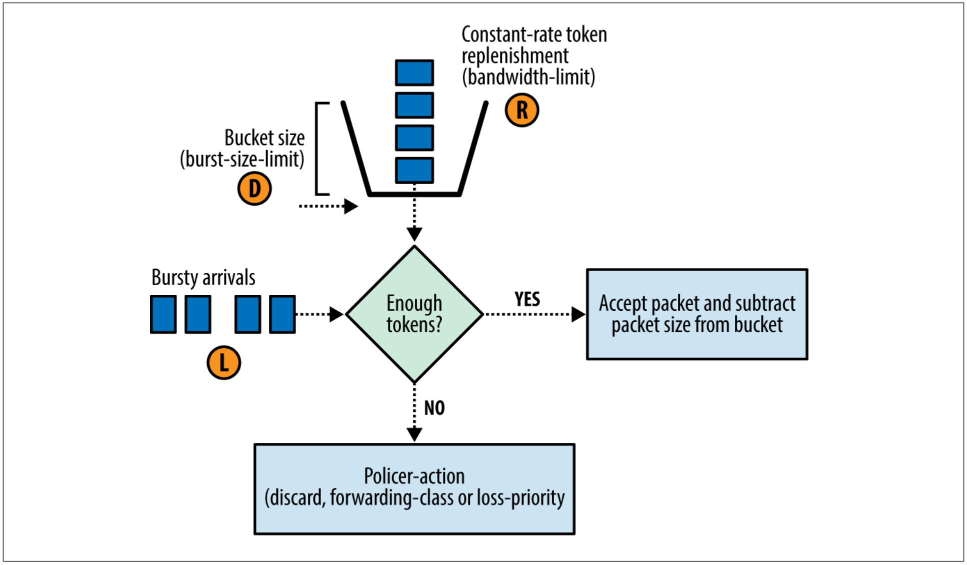

Constant-rate token replenishment (bandwidth-limit)
R
Bucket size (burst-size-limit)
D
Bursty arrivals
L
Enough tokens?
YES
Accept packet and subtract packet size from bucket
NO
Policer-action (discard, forwarding-class or loss-priority

Figure 3-3. Token Bucket Shaping Algorithm.

The figure shows how a token generator is used to produce tokens at a constant rate of R tokens per second, with the resulting tokens placed into a token bucket with a depth of D tokens. Generally, each token grants the ability to transmit a fixed number of bytes, so a greater bucket depth equates to a larger permitted burst size; when the token bucket fills, any newly generated tokens are discarded. This means that unused bandwidth from a previous cycle does not carry forward to grant increased usage credit in the current cycle. A classic case of use it or lose it!

The figure shows an unregulated stream of packets arriving at a packet queue with a maximum length of L, which is a function of the bucket size. As was the case with the simple leaky bucket algorithm, if the flow delivers more packets than the queue can store, the excess packets are discarded, a process that limits maximum burst rate as a function of queue depth. However, unlike the simple leaky bucket case, the token bucket's queue regulator has to consider a number of factors when deciding whether a packet of size P tokens can be forwarded into the network:

- When the token bucket is full, the packet is forwarded into the network and P tokens are removed from the bucket.
- When the token bucket is empty, the packet waits at the head of the queue until P tokens accumulate in the bucket. When P tokens eventually accrue, the packet is sent into the network and P tokens are removed from the bucket.
- When the token bucket is partially full and contains T tokens, the packet size must be factored. If P is less than or equal to T, P tokens are removed from the bucket

and the packet is sent. If P is greater than T, the packet must wait for the remaining P minus T tokens before it can be sent.

In this algorithm, the rate of the token generator defines the long-term average traffic rate while the depth of the token bucket defines the maximum burst size. As always, the length of the packet queue bounds the amount of delay that a packet can incur at the traffic shaper.

The token bucket and leaky bucket algorithms are both considered methods of shaping traffic because they regulate the long-term average transmission rate of a source. In contrast, the token bucket mechanism also supports the concept of a committed burst size. Burst support is important because most traffic is bursty by nature, and sending data in a burst will get it to its destination faster, assuming of course that downstream nodes are not chronically congested and can therefore handle the occasional traffic spikes. A leaky bucket shaper, with its constant traffic rate and lack of a burst, is more suited to networks that cannot tolerate any burst because they are always in a congested state, living at the very cusp of their buffer's capacity.

Policing

Traffic policing and shaping are often confused, perhaps because both can be based on a token bucket mechanism. Recall that a shaper either discards or delays traffic while waiting for sufficient token credit, and packets that leave a shaper are not marked. In contrast, a policer can perform discard actions as well, but it can also mark excess traffic. The policer marking can be used both in the local or downstream nodes to influence discard decisions during periods of congestion (sometimes called coloring or soft policing).

The policing function can be based on the token bucket algorithm, but now the packet queue is replaced with a metering section that accepts compliant traffic and either discards or marks excess traffic. The decision to drop a packet as opposed to marking it determines wither the policer is operating in a hard or soft model. Figure 3-4 illustrates a soft policer function.

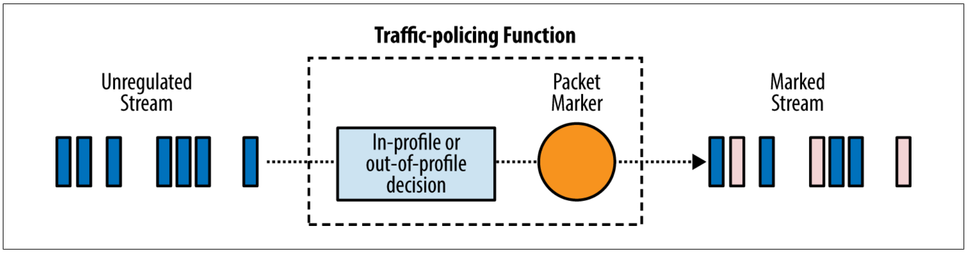

Traffic-policing Function
Unregulated Stream
Packet Marker
Marked Stream
In-profile or out-of-profile decision

Figure 3-4. Soft Policing Through Packet Marking.

As already noted, the decision to mark rather than to drop nonconformant traffic is the hallmark of a soft policer; in contrast, a hard policer immediately discards excess traffic.

A soft policer's traffic marking is used to influence drop behavior at each hop along the path to the packet's destination, where congestion management mechanisms like WRED are deployed to aggressively drop packets based on their color markings to help ensure that compliant traffic is never affected by excess traffic, and that excess traffic is the first to feel the pain during periods of congestion. Intelligent drop behavior is a critical component of maintaining separation between traffic forwarding classes, as demanded by the Differentiated Service CoS model.

When policing traffic, it's important to ensure that packet ordering is maintained within a flow. Simply marking a packet by increasing its drop precedence only raises the probability that a core router will drop the packet during periods of network congestion. Not receiving a packet is quite different than receiving packet number five before receiving packets one through four. Sequencing errors are particularly disastrous to real-time services like voice and video, which tend to perform better with simple loss as opposed to reordering.

In general, you avoid packet reordering by ensuring that all packets associated with a given flow are assigned to the same queue at each hop across the network. You should avoid using a policer to place out-of-profile traffic into a different forwarding class (or queue) because separating a single flow across multiple queues is a surefire recipe for a banquet of sequencing errors that will give all but the most robust of applications a bad case of heartburn.

Junos policers are just that: policers, not shapers. This is because policing takes a more flexible approach than traffic shaping; it provides the ability to burst, which both reduces delays and allows the network to deliver excess traffic when it has the capacity, in keeping with the principles of statistical multiplexing. It's important to note that a policer effectively functions like a shaper (at least at the macro view) when configured to operate in drop (hard) mode.

In the IETF's DiffServ parlance, policers and shapers perform the function of a "traffic conditioner." RFC 2475 "An Architecture for Differentiated Services" states that a traffic conditioner performs functions such as meters, markers, droppers, and shapers, and it may remark a traffic stream or may discard or shape packets to alter the temporal characteristics of the stream and bring it into compliance with a traffic profile. Given this definition, it's clear that a Junos policer can be classified as a DiffServ Traffic Conditioner, but as we are among friends we'll stick to the term policer.

Junos Policer Operation

Junos supports a single-rate two-color (srTC), single-rate three-color (srTCM), two-rate three-color (trTCM), and hierarchical policers. In addition, there are a number of different ways you can attach any given policer, the choice of which is a major factor in determining what traffic the policer sees. The details for the various policer types follow, but all policers share some common properties such as the use of a token bucket algorithm that limits traffic based on a bandwidth and burst size setting.

It's important to note that with all policer types, in-profile (conformant) traffic is always handed back to the calling term unmodified; the fate of excess or out-of-profile traffic is a function of the policer's configuration. For a traffic flow that conforms to the configured limits (categorized as green traffic), packets are marked with a packet loss priority (PLP) level of low and are typically allowed to pass through the interface unrestricted through the calling term's implicit accept action.

When a traffic flow exceeds the configured limits (classified as yellow or red traffic), the nonconforming packets are handled according to the actions configured for the policer. The action might be to discard the packet (hard policing), or the action might be to remark the packet (soft model), perhaps altering its forwarding class, or setting a specified PLP (also known as a color), or both, before the policer passes the traffic back to the calling term.

In the v11.4 Junos release, the Trio chipset on MX MPCs does not support explicit configuration of policer overhead. This feature is normally used in conjunction with interfaces that are rate limited using the `shaping-rate` statement configured under `[edit class-of-service traffic-control-profiles]`. When you define such a, added overhead for things like MPLS labels or VLAN tags can be factored into the shaped bandwidth using the `overhead-accounting` statement, which essentially allows you to add overhead bytes to the shaping rate so as to only shape user payload, when desired. If such an interface is also subjected to a policer, it's a good idea to add the same number of overhead bytes to match it to the shaper; stated differently, if you shape based on the payload, you should also police at the payload level whenever possible. Policer overhead for rate shaping is supported on I-Chip-based Dense Port Concentrators (DPCs) at this time.

Policer Parameters

The key parameters of Junos policers are the `bandwidth-limit` and `burst-size-limit` settings, as these combine to determine the average and peak bandwidth for conformant traffic.

Bandwidth

You set a policer bandwidth using the `bandwidth-limit` keyword. This parameter determines the rate at which tokens are added to the bucket, and therefore represents the highest average transmit (or receive) rate in bits per second (bps). When the traffic arrival rate exceeds the token replenishment rate, the traffic flow is no longer conforming to the bandwidth limit and the specified policer action is executed.

You specify the bandwidth limit as either an absolute number in bps or as a percentage value from 1 through 100. If a percentage value is specified, the effective bandwidth limit is calculated as a percentage of either the physical interface media rate or the logical interface's configured shaping rate.

When specifying bandwidth in absolute terms, the supported range is from 8,000 to 50,000,000,000 bps. The CLI supports human-friendly suffixes like `k`, `m`, or `g`, for kilo, megabits, or gigabits.

Burst Size

The maximum burst size is set with the `burst-size-limit` keyword. This parameter sets depth of the bucket, in bytes, and therefore the maximum amount of token credit that can accrue. Because tokens are needed to send traffic, this in turn limits the largest contiguous burst (in bytes) that is possible before nonconformance is declared.

> A properly dimensioned burst size is critical for proper operation. Setting the burst limit too low can cripple an application's performance. For example, consider that non-jumbo Ethernet frames have a MTU of 1,500, which means they can carry up to 1,500 bytes of upper layer protocol payload. During a file transfer, you might expect many such full-sized frames, which means that the *absolute minimum* acceptable burst size for this application is 1,500 bytes, assuming the policing is at Layer 3; for a Layer 2-based service, the additional 14 bytes (assuming no VLAN tagging or MPLS encapsulation) of frame overhead would have to be factored into the burst. In actuality, you will likely want to set the burst size several times higher than the absolute minimum, or else the policer will not be TCP-friendly and actual user throughput will likely be far below the policer bandwidth limit.

A Suggested Burst Size. As noted previously, setting too low a burst size can be disastrous to application throughput. On the other hand, setting the burst allowance too high can lead to a general lack of policing, perhaps causing havoc in other parts of the network when all that excess traffic results in congestion. Thus, the $64,000-dollar question is born: "What should I configure for a burst limit in my policers?"

The rule of thumb is that the burst size should never be lower than 10 times the interface's MTU, with the recommended setting for high-speed interfaces (i.e., Fast Ethernet and above) being equal to the amount of traffic that can be sent over the interface in five milliseconds. As an example, consider a 10 G Ethernet interface with a default 1,500 byte MTU. Given its bit rate, the interface can send 1.25 G Bytes per second, which means that in a given millisecond the interface can send 10 M bits; as a function of bytes, this equates to 1.25 MB per millisecond. Based on the previous guidelines, the minimum burst size for this interface is 15 KB/120 Kb (10 * 1500), while the recommended value, based on a 5 millisecond burst duration, comes out to be 6.25 MB/50 Mbps. Note the large delta between the minimum and recommended burst size settings. Given that MTU tends to be fixed, this ratio between minimum and recommended burst size grows with increasing interface speed. While a UDP or traffic generator stream may see similar results with each over some period of time, a TCP-based application will likely perform quite poorly with the lower setting. This is because TCP interprets loss to mean congestion, so each policer discard leads the sender to reduce its congestion window and to initiate slow start procedures, with the result being that actual user throughput can fall well below the policer bandwidth limit.

Policer Actions

Once the policer limits have been configured, the action taken if a packet exceeds the policer must be chosen. There are two types of policing available: "soft" policing and "hard" policing. Hard policing specifies that the packet will be dropped if it exceeds the policer's traffic profile. Soft policing simply marks and/or reclassifies the packet, both of which are actions intended to increase the probability of the packet being dropped during times of congestion. In this context, marking refers to modifying a packet's packet loss priority (PLP), which is an internal construct used to influence the local node's discard and outgoing header rewrite actions, with the latter used to convey loss priority to a downstream node. This process is also known as coloring, especially when the router supports more than two loss priorities. These concepts are further examined in the Class of Service chapter.

Basic Policer Example

In Junos, the most basic style of policer is technically referred to as a single-rate two-color (srTC) policer. You configure this type of policer at the `[edit firewall policer]` hierarchy. This policer classifies traffic into two groups using only the `bandwidth-limit` and `burst-size-limit` parameters. Assuming a soft model, the traffic that conforms to the bandwidth limit or the peak burst size is marked green (low PLP) while traffic in excess of both the bandwidth limit and the peak burst size is marked red (high PLP) and possibly assigned to a different FC/queue.

In some cases, a discard action for excess traffic may be desired. Hard policing is often seen for ingress traffic contract enforcement or with traffic that is associated with a strict-high (SH) scheduling priority. This is because without the use of `rate-limit` to cap bandwidth consumption, an SH scheduler is not subjected to a transmit rate and would otherwise only be limited by interface bandwidth (or the shaping rate when in effect) and as such excess SH traffic can easily starve other traffic classes that are within their bandwidth limits. Chapter 5 provides details on Junos schedulers and relative priorities.

A sample srTC policer is shown:

```
policer simple {
    if-exceeding {
        bandwidth-limit 50m;
        burst-size-limit 15k;
    }
    then discard;
}
```

Given the previous discussion, there is not much to say here, except perhaps that the `discard` action makes this a hard policer example, and that the burst size, at 15 KB, appears to be set based on the minimum recommendation of 10 times the default Ethernet MTU of 1,500 bytes. To be useful, such a policer must be applied in the packet forwarding path. This can be done via a firewall filter or through direct application to an interface, as described in the following.

The basic single-rate two-color policer is most useful for metering traffic at a port (IFD) level. However, you can also apply this style policer at the logical interface (ifl) level or as part of a MF classifier. There are two main modes of operating for this style of policer.

Bandwidth Policer

A bandwidth policer is simply a single-rate two-color policer that is defined using a bandwidth limit specified as a percentage value rather than as an absolute number of bits per second. When you apply the policer (as an interface policer or as a firewall filter policer) to a logical interface at the protocol family level, the actual bit rate is computed based on the physical device speed and the specified bandwidth percentage.

Logical Bandwidth Policer

A logical bandwidth policer is when the effective bandwidth limit is calculated based on the logical interface configured shaping rate. You can apply as a firewall filter policer only, and the firewall filter must be configured as an interface-specific filter. You create this style of policer by including the `logical-bandwidth-policer` statement. The resulting bit rate is computed based on any shaping rate applied to the interface, rather than its actual media speed. Shaping is covered in detail in Chapter 5.

Cascaded Policers

With Junos, you can use filter-evoked policing along with the `next-term` flow-control operator to achieve cascaded policing. A cascaded policer allows you to police the same stream more than once: first at some aggregate rate, and then again at a lesser rate for some specific subset of the traffic. This concept is shown in Figure 3-5.

Some refer to this approach as hierarchical or nested policers. The term "cascaded" is used here to differentiate this concept from a Hierarchical Policer, as described in the following, and from firewall nesting, where you call a filter from another filter.

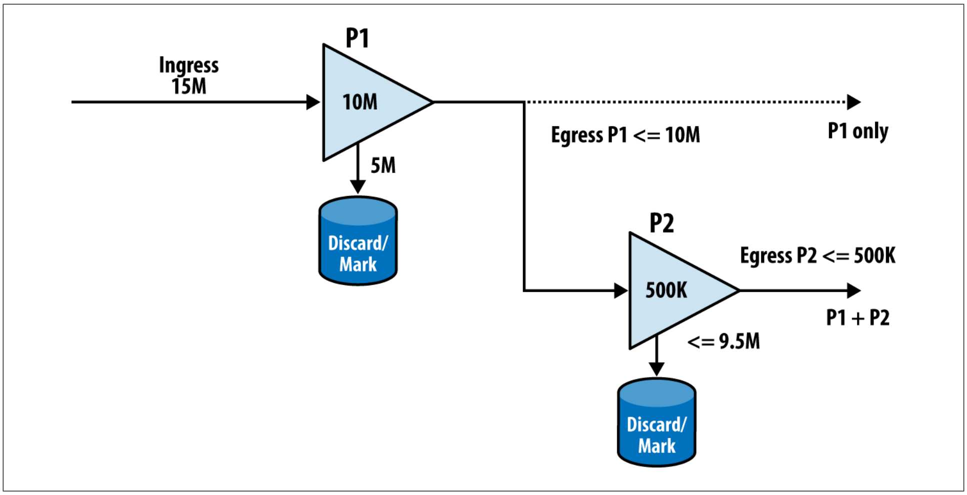


Figure 3-5. Cascaded Policers, A Poor man's Hierarchical Policer.

For the sake of brevity, the policers in this example are shown with a bandwidth limit only. A match-all filter term is used to direct all traffic to the first stage policer (P1) on the left, where it is limited to 10 Mbps. The term used to evoke the policer must use the next-term flow controller modifier to avoid implicit acceptance of in-profile aggregate traffic.

The cascading comes to play when a subset of P1's output is then subjected to a second level of policing at P2. A second filter, or filter term, is used to match the subset of the aggregate flow that is to receive additional policing, which in this example limits that traffic to 500 Kbps. The key here is that P1 > P2, and as such P1 limits the maximum stream rate to 10 Mbps or less. This 10 Mbps could be only P1 traffic, such as would occur when none of the traffic is subjected to the second policer, or it could be comprised of as much as 9.5 Mbps of P1 + 500 K of P2 in the event that 500 K of P2 traffic is matched. In contrast, if *only* P2 traffic is present then the maximum output rate is the 500 K associated with P2.

A sample Junos configuration for a cascaded policer is shown; the example includes the mandatory burst parameters that were omitted from the figure:

```
user@R1# show firewall
policer P1 {
    if-exceeding {
        bandwidth-limit 10m;
```

```
            burst-size-limit 50k;
        }
        then discard;
    }
    policer P2 {
        if-exceeding {
            bandwidth-limit 500k;
            burst-size-limit 50k;
        }
        then discard;
    }
```

A corresponding filter is defined to call the policers for matching traffic:

```
    firewall {
        family inet {
            filter cascaded_policing_example {
                interface-specific;
                term limit_aggregate_P1 {
                    then policer P1;
                     then next-term;
                }
                term limit_icmp_P2 {
                    from {
                        protocol icmp;
                    }
                    then policer P2;
                }
                term catch-all {
                    then accept;
                }
            }
        }
    }
```

The example filter named `cascaded_policing_example` consists of three terms. The first and last match all traffic given their lack of a `from` match condition. The effect of the `limit_aggregate_P1` term is to direct all traffic to the `P1` policer, which limits the aggregate to 10 Mbps. Note that the outright acceptance of in-profile traffic is prevented here through use of the `next-term` action modifier. The next-term action is a key component in this filter-based nested policer example; without it, no traffic can ever make it to the `limit_icmp_P2` term, a condition that clearly defeats the cascaded policing plan. The second term only matches on ICMP traffic, with matching traffic directed to the `P2` policer, where it's rate limited to the lesser value of 500 Kbps.

At this stage, the reader should know the purpose of the final `catch-all` term. Recall that without this explicit accept-all term *only* ICMP traffic (limited to 500 Kbps) can be sent, as all other traffic would be subjected to the implicit deny-all function.

Single and Two-Rate Three-Color Policers

MX routers also support single-rate three-color narker (srTCM) and two-rate three-color marker (trTCM) style policers, as described in RFCs 2697 and 2698, respectively. These policers enhance the basic single-rate two-color marking type previously discussed by adding a third packet coloring (yellow) and increased control over bursting.

The main difference between a single-rate and a two-rate policer is that the single-rate policer allows bursts of traffic for short periods, while the two-rate policer allows more sustained bursts of traffic. Single-rate policing is implemented using a dual token bucket model that links the buckets so that overflow from the first is used to fill the second, so that periods of relatively low traffic must occur between traffic bursts to allow both buckets to refill. Two-rate policing is implemented using a dual token-bucket model where both buckets are filled independently, as described subsequently.

Another way of looking at their differences is to consider that operationally, a srTCM provides moderate allowances for short periods of traffic that exceed the committed burst size, whereas a trTCM accommodates *sustained* periods of traffic that exceeds the committed bandwidth limit or burst size.

The drawback to a single-rate policer is that network operators tend to provision smaller CIRs (or fewer customers) to ensure they can meet simultaneous CIR rates by all customers, which is a rare event and usually results in bandwidth underutilization during the far more common periods of low activity. This shortcoming resulted in the concept of a dual-rate traffic contract, where control is provided over two sending rates, with only the lesser one guaranteed. A two-rate contract first became popular in Frame Relay networks; as an example, consider the extremely popular case of a 0 CIR service with excess burst rate (Be) set to the port speed (access rate). Here, the network guarantees nothing, and so can sell services to as many customers as they like without technically ever overbooking their capacity. But, users can still burst to ports speed with the only drawback being all their traffic is marked as Discard Eligible (DE). Given the high capacity of modern networks, and the statistically bursty nature of IP applications, most customers find this type of service to be both reliable and cost-effective, often choosing it over a higher CIR guarantee that also comes with higher rates and DLCI limits based on port speed, as a function of the aggregate CIR not being allowed to exceed port speed.

TCM Traffic Parameters

Recall that a single-rate two-color marker-style policer incorporated two traffic parameters to determine if policed traffic is in or out of the policer profiler; namely, the bandwidth and burst-size limits. TCM-style policers must support additional traffic parameters to perform their job. While there are some new names, always remember that in a Junos policer bandwidth is always measured in bits and burst is always measured in bytes.

Single-Rate Traffic Parameters. A srTCM policer is defined by three traffic parameters: the CIR, CBS, and EBS.

Committed Information Rate

The committed information rate (CIR) parameter defines the long-term or average guaranteed bandwidth of a service as measured in bps by controlling the rate at which tokens are added to the CBS bucket. A traffic flow at or below the CIR is marked green. When traffic is below the CIR, the unused bandwidth capacity accumulates in the first token bucket, up to the maximum defined by the CBS parameter. Any additional unused tokens can then overflow into the second bucket that controls excess bursting.

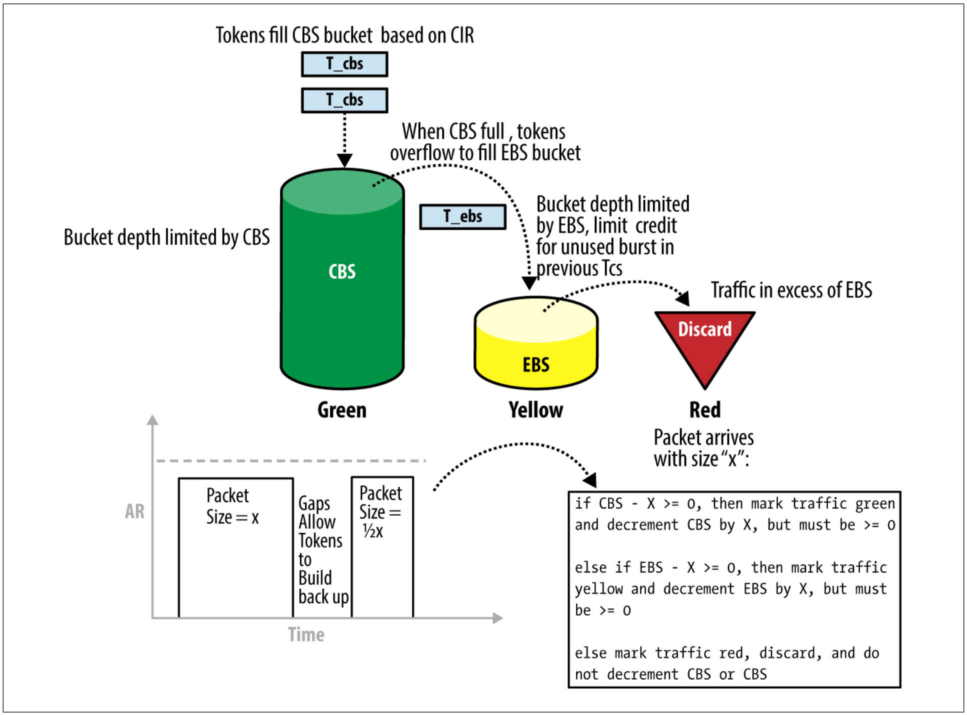
Committed Burst Size

The committed burst size (CBS) defines the maximum number of bytes that can be accumulated in the first token bucket, and therefore the maximum size (in bytes) of a committed burst. A burst of traffic can temporarily exceed the CIR and still be categorized as green, provided that sufficient bandwidth capacity has been allowed to build in the first bucket due to a previous period in which traffic was below the CIR.

Excess Burst Size

The srTCM policer configuration includes a second burst parameter called the excess burst size (EBS). This parameter defines the maximum number of token credits that can overflow from the CBS bucket to accumulate in the EBS bucket. The depth of this bucket determines how many credits you build up for unused committed burst in previous intervals, which can now be applied to traffic that's beyond the CBS in the current interval. Larger values allow the user to reclaim more unused intervals for application against excess traffic in a current interval, and so allow for a longer burst.

Figure 3-6 shows how the srTCM policer uses these parameters to meter traffic.


Tokens fill CBS bucket based on CIR
T_cbs
T_cbs
When CBS full, tokens overflow to fill EBS bucket
Bucket depth limited by CBS
CBS
T_ebs
Bucket depth limited by EBS, limit credit for unused burst in previous Tcs
Traffic in excess of EBS
Discard
EBS
Green
Yellow
Red
Packet arrives with size "x":
AR
Packet Size = x
Gaps Allow Tokens to Build back up
Packet Size = ½x
Time

```
if CBS - X >= 0, then mark traffic green
and decrement CBS by X, but must be >= 0

else if EBS - X >= 0, then mark traffic
yellow and decrement EBS by X, but must
be >= 0

else mark traffic red, discard, and do
not decrement CBS or CBS
```

Figure 3-6. The srTCM Policer.

Initially, both the committed and excess buckets are filled to their capacity. Their depth is a function of the configured CBS and EBS parameters, respectively. Token credits are added to the CBS bucket based on the configured CIR. If the CBS bucket fills, overflow tokens begin to fill the EBS bucket until it too is full, in which case no additional tokens accumulate. Their combined token capacity sets a hard limit on how much traffic can be sent in a contiguous burst before the discard action associated with red traffic is initiated.

Packets are sent at the Access Rate (AR) of the interface, also known as port speed, which normally far exceeds the CIR. Thus, it's the length of a burst, not its arrival rate (in bps), that determines if traffic exceeds one or both buckets. When a packet of size X arrives, the srTCM algorithm first tries to send it as CBS by comparing X to the number of token credits in the CBS bucket. If X is smaller, the CBS bucket is decremented by X, to a minimum of 0, and the traffic is marked green.

If insufficient token credit is found in the CBS bucket, the traffic is next compared to the credit in the EBS bucket. Recall that EBS provides credit for excess traffic in the current interval as a function of having sent less than you could have in a previous time period. If X is less than or equal to the credits in the EBS bucket, the packet is marked yellow and the EBS token reserve is decremented by X. If X is larger than the credit found in both buckets, the packet is marked red; no changes are made to the token credits in either bucket, thereby allowing them to continue accumulating credits based on the CIR.

In effect, the EBS improves "fairness" to compensate for cases where a link has been idle for some period of time just before a large burst of traffic arrives. Without tolerance for some amount of excess traffic in such a case, the user is limited to his or her CBS despite the previous idle period. By allowing the customer to accumulate up to EBS extra bytes, the customer is compensated for idle times by being able to send CBS + EBS traffic. It's important to note that the result is that the long-term average rate still remains equal to CIR. Thus, a srTCM is said to be best suited to the support of traffic that exhibits only occasional bursting.

Two-Rate Traffic Parameters. A two-rate traffic contract is defined by four parameters, namely, the CIR, CBS, PIR, and PBS.

Committed Information Rate
: The trTCM's CIR parameter functions the same as in the srTCM case described previously in the srTCM case.

Committed Burst Size
: The trTCM's CBS parameter functions the same as in the srTCM case described previously for the srTCM.

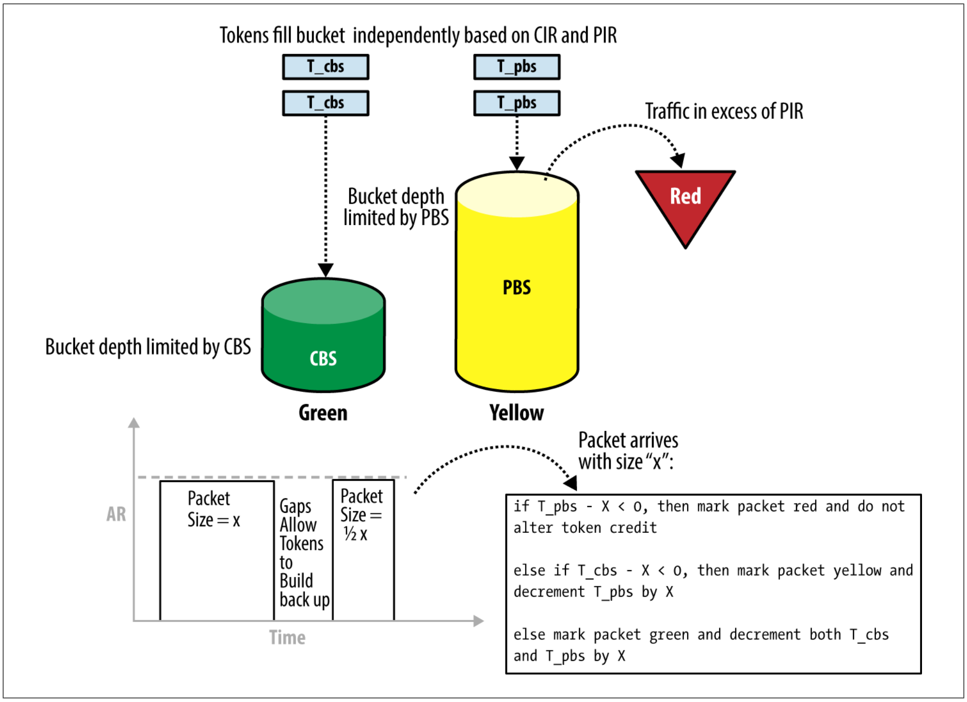
Peak Information Rate
: A trTCM policer uses a second rate limit parameter called the Peak Information Rate (PIR). Like the CIR, this parameter defines the maximum average sending rate over a one-second period. Traffic bursts that exceed CIR but remain under PIR are allowed in the network, but are marked yellow to signal their increased drop probability.

Peak Burst Size
: A trTCM policer uses a second burst size parameter called the Peak Burst Size (PBS). This parameter defines the maximum number of bytes of unused peak bandwidth capacity that the second token bucket can accumulate. During periods of relatively little peak traffic (the average traffic rate does not exceed the PIR), unused peak bandwidth capacity accumulates in the second token bucket, but only up to the maximum number of bytes specified by the PBS. This parameter places a limit on the maximum length of a burst before the PIR is violated, and the traffic is marked red and can be subjected to immediate discard when so configured.

The use of two independent buckets combined with the additional parameters needed for control of PIR independently of CIR facilitates support of sustained bursting when compared to a srTCM. Figure 3-7 details the operation of a trTCM.

As before, both buckets are initially filled to their capacity, and each bucket's depth is a function of the configured CBS and PBR, respectively. However, in this case token credits are added to both buckets independently based on the configured CIR and PIR; there is no overflow from the first to fill the second as was the case with the srTCM. In fact, when both buckets are full no additional tokens can accumulate anywhere, meaning there is less fairness in the face of bursty traffic with regards to unused capacity and the current intervals committed burst.

When a packet of size X arrives, the trTCM algorithm first determines if it exceeds the PIR by comparing its size to the token reserve in the PBS bucket. If X exceeds the current T_pbstoken capacity, the traffic is marked red; no token counters are decremented for red traffic as it was never sent.


Tokens fill bucket independently based on CIR and PIR
T_cbs
T_cbs
T_pbs
T_pbs
Traffic in excess of PIR
Bucket depth limited by PBS
Red
PBS
Bucket depth limited by CBS
CBS
Green
Yellow
Packet arrives with size "x":
AR
Packet Size = x
Gaps Allow Tokens to Build back up
Packet Size = ½ x
Time

```
if T_pbs - X < 0, then mark packet red and do not
alter token credit

else if T_cbs - X < 0, then mark packet yellow and
decrement T_pbs by X

else mark packet green and decrement both T_cbs
and T_pbs by X
```

Figure 3-7. The trTCM Policer.

When a packet does not exceed the T_pbsreserve, the next step is to determine green or yellow status by comparing size X to the T_cbs reserve in the CBS bucket. If X is larger than the current CBS capacity, the traffic is marked yellow and the corresponding number of PBS tokens are deleted from the PBS bucket. If the current packet size does not exceed the CBS bucket's token reserve, the traffic is marked green and the corresponding number of tokens is deleted from *both* the CBS and PBS buckets. The result is that yellow traffic counts only against the peak burst while green traffic counts against both committed and peak burst.

With a trTCM there is no ability to store unused capacity, which is to say that unused tokens simply overflow from one or both buckets. As such, there is no "fairness" mechanism in trTCM with regard to gaining extra capacity based on unused capacity in a previous interval. Instead, every incoming packet is compared to the current token capacity of both the CBS and PBS buckets to compare the current traffic flow to the two defined rates. This makes the trTCM better suited for traffic that is chronically bursty, as the long-term average rate is limited by the PIR, rather than by whether all capacity was used in the last measurement period. Recall that with the srTCM, bursting is really only possible as a function of not sending at the CIR in a previous interval. In contrast, the use of two independent buckets in trTCM means the user can send PIR in each and every interval. That said, only the CIR traffic is guaranteed, as traffic above the CIR may be discarded during periods of congestion due to its yellow marking.

Color Modes for Three-Color Policers

Both single-rate and two-rate three-color policers can operate in either a color-blind or color-aware mode.

Color-Blind Mode

In color-blind mode, the three-color policer assumes that all packets examined have not been previously marked or metered. If you configure a three-color policer to be color-blind, the policer ignores preexisting color markings that might have been determined via a BA classifier or set for a packet by another traffic policer (i.e., an ingress policer or as part of a nested policer design).

Color-Aware Mode

In color-aware mode, the three-color policer assumes that all packets examined have been previously marked or metered and any preexisting color markings are used in determining the appropriate policing action for the packet.

It should be noted that the three-color policer can *only* increase the packet loss priority (PLP) of a packet (i.e., make it more likely to be discarded). For example, if a color-aware three-color policer meters a packet with a medium PLP marking, it can raise the PLP level to high but cannot reduce the PLP level to low. This ensures that a packet's CoS is never increased by being subjected to additional levels of policing.

If a previous two-rate policer has marked a packet as yellow (loss priority medium-low), the color-aware policer takes this yellow marking into account when determining the appropriate policing action. In color-aware policing, the yellow packet would never receive the action associated with either the green packets or red packets. As a result, tokens for violating packets are never taken from the metering buckets for compliant traffic, or stated differently, tokens of a given color are always used for traffic of the same color. Therefore, the total volume of green traffic should never be lower than the configured CIR/CBS permits.

Configure Single-Rate Three-Color Policers

You configure a srTCM at the `[edit firewall three-color-policer]` hierarchy. You must also include the `single-rate (color-aware | color-blind)` statement, in addition to specifying the committed-information-rate, committed-burst-size, and excess-burst-size parameters. In addition, to evoke such a policer from a filter you must do so with a then three-color-policer action, which requires that you also specify the related single-rate policer by its name.

A srTCM policer takes the following form:

```
{master}[edit firewall three-color-policer test_sr_tcm]
jnpr@R1-RE0# show
action {
    loss-priority high then discard;
}
single-rate {
    committed-information-rate 1m;
    committed-burst-size 64k;
    excess-burst-size 128k;
}
```

An srTCM is evoked either through direct application to an interface, or via a filter call. To call from a filter, include the three-color-policer and single-rate statements as shown:

```
filter limit_ftp {
    term 1 {
        from {
            protocol tcp;
            port [ ftp ftp-data ];
        }
        then {
            three-color-policer {
                single-rate test_sr_tcm;
            }
        }
    }
    term 2 {
        then accept;
    }
}
```

srTCM Nonconformance. An srTCM policer marks traffic yellow when its average rate exceeds the CIR (and therefore the available bandwidth capacity accumulated in the first bucket) when there is sufficient unused bandwidth capacity available in the second token bucket. Packets in a yellow flow are implicitly marked with medium-high PLP and then passed through the interface.

A traffic flow is categorized red when the average rate exceeds the CIR *and* the available bandwidth capacity accumulated in the second bucket. Packets in a red flow are the first to be discarded in times of congestion, and based on configuration can be discarded immediately, independent of congestion state.

Configure Two-Rate Three-Color Policers

A trTCM policer is also defined at the [edit firewall three-color-policer] hierarchy. However, you now include the two-rate (color-aware | color-blind) statement, in addition to specifying the committed-information-rate, committed-burst-size, peak-information-rate, and peak-burst-size parameters. In addition, when you evoke such a policer from a filter, you must do so with a then three-color-policer action, which requires that you also specify the related two-rate policer by its name.

A typical trTCM policer takes the following form, and like the srTCM, is normally evoked via filter:

```
{master}[edit firewall]
jnpr@R1-RE0# show
three-color-policer test_tr_tcm {
    action {
        loss-priority high then discard;
    }
    two-rate {
        committed-information-rate 1m;
        committed-burst-size 64k;
        peak-information-rate 2m;
        peak-burst-size 128k;
    }
}
filter limit_ftp {
    term 1 {
        from {
            protocol tcp;
            port [ ftp ftp-data ];
        }
        then {
            three-color-policer {
                two-rate test_tr_tcm;
            }
        }
    }
    term 2 {
        then accept;
    }
}
```

Note that the filter makes use of the three-color-policer, as did the srTCM, but is now combined with a two-rate keyword to call for the instantiation of a trTCM-style policer.

Two-rate policing is implemented using a dual token-bucket model, which allows bursts of traffic for longer periods. The trTCM is useful for ingress policing of a service, where a peak rate needs to be enforced separately from a committed rate. As stated previously, a trTCM policer provides moderate allowances for *sustained* periods of traffic that exceed the committed bandwidth limit and burst size parameters.

trTCM Nonconformance. A trTCM marks a packet yellow when it exceeds the CIR (based on the committed bandwidth capacity held in the first token bucket) while still conforming to the PIR. Packets in a yellow flow are implicitly marked with medium-high PLP and then passed through the interface. A packet is colored red when it exceeds the PIR and the available peak bandwidth capacity of the second token bucket (as defined by the PBS parameter). Packets in a red flow are not discarded automatically unless you configure a discard action in the policer.

Hierarchical Policers

Junos supports the notion of a hierarchical policer on certain platforms. This type of policer functions by applying different policing actions based on traffic classification as either EF/premium or other/aggregate. Support for hierarchical policing in Trio-based MX platforms began in release v11.4R1. This style of policer is well suited to a Service Provider edge application that needs to support a large number of users on a single IFD, when the goal is to perform aggregate policing for the all the traffic as well as separate policing for the premium traffic from all users on the subscriber-facing interface.

You use a hierarchical policer to rate limit ingress traffic at Layer 2 either at the logical interface (IFL) level or at the interface (IFD) level. When applied to an IFL (with at least one protocol family), a hierarchical policer rate limits ingress Layer 2 traffic for all protocol families configured on that IFL. In contrast, the hierarchical policer affects all protocols on all IFLs when applied at the interface (IFD) level. You cannot apply a hierarchical policer to egress traffic. Furthermore, it only applies to Layer 3 traffic, or on a specific protocol family basis when multiple families share the interface, or logical unit, for IFD- versus IFL-level applications, respectively.

> Trio-based MX platforms currently support hierarchical policers at the IFL level under family `inet`, `inet6`, or `mpls` only. Application at the IFL requires that at least one protocol family be configured.

For hierarchical policers to work, ingress traffic must be correctly classified into premium and nonpremium buckets. Therefore, it's assumed that a Behavior Aggregate (BA)- or Multi-Field (MF)-based classification is performed prior to any hierarchical policing.

You specify two policer rates when you configure a hierarchical policer: a premium rate for EF traffic and an aggregate policer rate for all non-EF. The policers then function in a hierarchical manner.

Premium Policer
: You configure the premium policer with a guaranteed bandwidth and a corresponding burst size for high-priority EF traffic only. EF traffic is categorized as nonconforming when its average arrival rate exceeds the guaranteed bandwidth and the average burst size exceeds the premium burst size limit. A premium policer

must discard all nonconfirming traffic. This is in keeping with EF normally being scheduled with Strict High (SH) priority, which in turn necessitates a rate limit utilizing hard policing at ingress to prevent starvation of other forwarding classes (FCs).

Aggregate Policer

The aggregate policer is configured with an aggregate bandwidth that is sized to accommodate both high-priority EF traffic, up to its guaranteed bandwidth, and normal-priority non-EF traffic. In addition, you also set a supported burst size for the non-EF aggregate traffic only.

Non-EF traffic is categorized as nonconforming when its average arrival rate exceeds the amount of aggregate bandwidth not currently consumed by EF traffic and its average burst size exceeds the burst size limit defined in the aggregate policer. The configurable actions for nonconforming traffic in an aggregate policer are discard, assign a forwarding class, or assign a PLP level; currently, you cannot combine multiple actions for nonconformant traffic.

In operation, EF traffic is guaranteed the bandwidth specified as the premium bandwidth limit, while non-EF traffic is rate limited to the amount of aggregate bandwidth not currently consumed by the EF traffic. Non-EF traffic is rate limited to the entire aggregate bandwidth only when no EF traffic is present.

> You must configure the bandwidth limit of the premium policer at or below the bandwidth limit of the aggregate policer. If both the premium and aggregate rates are equal, non-EF traffic passes through the interface unrestricted *only* while no EF traffic arrives at the interface.

Hierarchical Policer Example

You configure a hierarchical policer at the `[edit firewall hierarchical-policer]` hierarchy. In addition to the policer itself, you may need to modify your CoS configuration at `[edit class-of-service]` hierarchy; some CoS configuration is needed because this style of policer must be able to recognize premium/EF from other traffic classes.

A sample hierarchal policer is shown:

```
{master}[edit]
jnpr@R1-RE0# show firewall
hierarchical-policer test_hierarchical-policer {
    aggregate {
        if-exceeding {
            bandwidth-limit 10m;
            burst-size-limit 100k;
        }
        then {
            loss-priority high;
        }
    }
    premium {
        if-exceeding {
            bandwidth-limit 2m;
            burst-size-limit 50k;
        }
        then {
            discard;
        }
    }
}
```

In this example, the policer for premium has its bandwidth limit set to 2 Mbps and its burst size limit set to 50 k bytes; the nonconforming action is set to discard packets, which is the only option allowed for the premium policer. The aggregate policer has its bandwidth limit set to 10 Mbps, which is higher than the premium policer bandwidth limit, as required for proper operation. The burst size limit for aggregate traffic is set to 100 k bytes, and in this example, the nonconforming action set to mark high PLP. Note again that the aggregate policer can discard the packet, change the loss priority, or change the forwarding class; currently, multiple action modifiers for out of profile aggregate traffic are not supported, but this capability is supported by the underlying chipset.

The hierarchical policer must be applied to an interface to take effect. As R1 is an MX router with a Trio-based PFE, you must apply the hierarchical policer at the IFL level, under either the `inet` or `inet6` family. Despite its application under the `inet` family in this example, once applied it will police all families that share the same ifl.

```
jnpr@R1-RE0# show interfaces xe-2/1/1
unit 0 {
    family inet {
        input-hierarchical-policer test_hierarchical-policer;
        address 192.168.0.2/30;
    }
}
```

As noted previously, you must also define CoS forwarding classes to include the designation of which FC is considered premium (By default this is the FC associated with EF traffic).

```
{master}[edit]
jnpr@R1-RE0# show class-of-service forwarding-classes
class fc0 queue-num 0 priority low policing-priority normal;
class fc1 queue-num 1 priority low policing-priority normal;
class fc2 queue-num 2 priority high policing-priority premium;
class fc3 queue-num 3 priority low policing-priority normal;
```

And, don't forget that you must also apply a BA classifier, as in this example, or an MF classifier via a firewall filter, to the interface to ensure that traffic is properly classified before it's subjected to the hierarchical policer. By defaultn an IPv4-enabled interface will have the `ipprec-compatibility` classifier in effect and that classifier only supports 2 FCs, FC0n and FC3. Here a DSCP-based BA is applied using the default DSCP code point mappings:

```
{master}[edit]
jnpr@R1-RE0# show class-of-service interfaces
xe-2/1/1 {
    unit 0 {
        classifiers {
            dscp default;
        }
    }
}
```

The effect of this configuration is that EF traffic is guaranteed a bandwidth of 2 Mbps. Bursts of EF traffic that arrive at rates above 2 Mbps can also pass through the interface, provided sufficient tokens are available in the 50 k byte burst bucket. When no tokens are available, EF traffic is rate limited using the discard action associated with the premium policer.

Non-EF traffic is metered to a bandwidth limit that ranges between 8 Mbps and 10 Mbps, depending on the average arrival rate of the EF traffic. Bursts of non-EF are allowed through the interface, provided sufficient tokens are available in the 100 K bandwidth bucket. Aggregate traffic in excess of the currently allowed bandwidth or burst size are rate limited using the action specified for the aggregate policer, which in this example is to set a high PLP.

You can apply a hierarchal policer as a physical interface policer by including the `physical-interface-policer` statement and then applying the hierarchical policer to the interface under a supported protocol family, as described in the section about applying policers.

Applying Filters and Policers

Once your firewall filters and policers are defined, you must apply them so they can take effect. Generally speaking, a filter is applied at the IFL level for a given family or at the IFL itself when using any family. In contrast, a policer can be applied to an IFL either directly or indirectly via a filter that calls the policer function. You can also apply a policer to the entire IFD in what is referred to as a *physical interface policer*.

This section details options for applying filters and policers on an MX router.

Filter Application Points

Firewall filters can be applied in a number of different locations along a packet's processing path through the router; it's critical to understand these options and their implications when you deploy a filter to ensure expected behavior.

The reader will recall that Chapter 1 provides a detailed description of packet flow through a Trio-based MX router. Figure 3-8 details filter and policer application points for a Trio-based MX router.

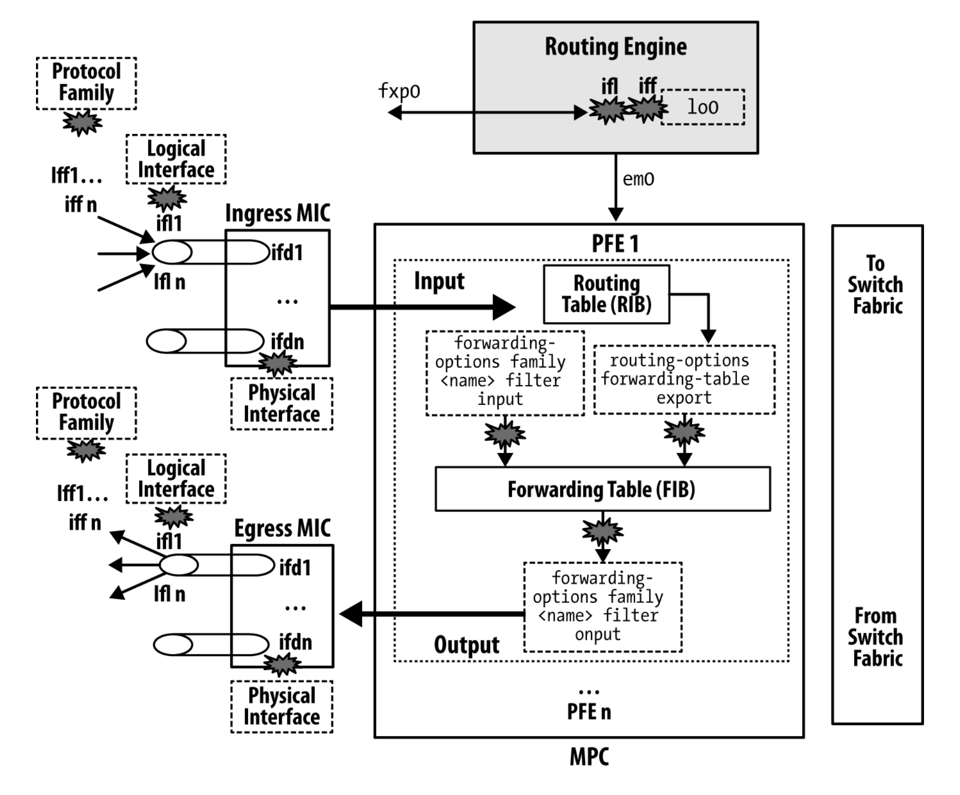


Figure 3-8. Trio PFE Filter Application Points.

Loopback Filters and RE Protection

The top of the figure shows how an lo0 filter is applied to filter traffic moving to or from the RE. An lo0 filter does not affect transit traffic directly. You typically apply an lo0 filter in the input direction to filter incoming remote access and routing protocol traffic that is received on the OOB or via a PFE network interface.

While less common, you can apply in the output direction to filter traffic originated by the local RE. But output lo0 filters are rarely used for security—after all, you normally trust your own internal devices to send nonmalicious traffic. Instead, output lo0 filters are typically used to alter the default CoS marking and queuing behavior for locally generated traffic. This topic is explored in detail later in Chapter 5.

At the time of the writing of this book, you cannot apply filters for Layer 2 families to the lo0 interface, but you can apply a `family any` filter to the IFL level, and family `inet4` or `inet6` filters at the respective IFF levels.

VRF instances with an lo0.x interface that has an input filter applied perform only the actions specified in the instance filter chain. If no lo0 unit is provisioned in the instance (and therefore there is no instance-specific input lo0 filter), all host-bound traffic from the instance is subjected to the main instance input lo0 filter (assuming one is defined) before being processed by the RE.

Input Interface Filters

The middle-left portion of Figure 3-8 shows ingress filter application points. Input filters can be applied in a number of different locations, starting with the ingress interface level (IFD), and moving up to the IFL and ultimately the IFF levels. Selecting where to apply a filter is a function of the overall goal and the protocols being filtered. For example, use a `family any` filter at the logical unit (IFL) level when the goal is to filter various protocol families using generic criteria such as packet size or forwarding class. In contrast, apply a filter at the protocol family (IFF) level when you wish to filter on the specific fields of a given protocol. The figure shows a physical interface filter as operating at the physical interface (IFD) level. While the IFD-level operation shown in the figure is accurate, it should be noted that you *cannot directly* attach a physical interface filter (or policer) to an IFD. Physical interface filters/policers are detailed in a later section to make their configuration and use clear.

When a packet is accepted by an input filter, it's directed to the route lookup function in the PFE. The packet is then queued for transmission to any receivers local to the current PFE and over the switch fabric to reach any remote receivers attached to other PFEs, even when those other PFEs might be housed on the same MPC.

Output Interface Filters

Egress traffic is shown in the lower left-hand portion of Figure 3-8, with traffic from remote PFEs being received via the switch fabric interface. The egress PFE performs its own packet lookups to determine if the traffic should be sent to any output filters. Output filters can be applied at the IFD-, IFL-, or IFF-level application points, just like their input filter counterparts. When accepted by an output filter, the packet is allowed to be pulled from shared memory and transmitted out the egress interface.

You can apply the same or different filters, in both the input and output directions, simultaneously.

Unlike lo0 filters that affect local host traffic only, a filter that is applied to a transient network interface (i.e., one housed in the PFE) can affect both transit and RE-bound traffic.

Junos filters follow a “prudent” security model, which is to say that by default they end in an implicit deny-all rule that drops all traffic that is not otherwise explicitly permitted by a previous filter term. It’s a good idea to make use of `commit confirmed` when activating a new filter application, especially so if you are remote to the router being modified. This way if you lose access to the router, or something really bad happens, the change will be automatically rolled back (and committed) to take you back where you were before the last commit.

Aggregate or Interface Specific

The same filter can be applied to multiple interfaces at the same time. By default on MX routers, these filters will sum (or aggregate) their counters and policing actions when those interfaces share a PFE. You can override this behavior and make each counter or policer function specific to each interface application by including the `interface-specific` keyword in the filter. If you apply such a filter to multiple interfaces, any counter or policer actions act on the traffic stream by entering or exiting each individual interface, regardless of the sum of traffic on the multiple interfaces.

When you define an interface-specific filter, you must limit the filter name to 52 bytes or less. This is because firewall filter names are restricted to 64 bytes in length and interface-specific filters have the specific interface name appended to them to differentiate their counters and policer actions. If the automatically generated filter instance name exceeds this maximum length, the system may reject the filter’s instance name.

Filter Chaining

You can apply as many as 16 separate filters to a single logical interface as a filter list. Similar to applying a policy chain via the assignment of multiple routing policies to a particular protocol, the ordering of the filters is significant, as each packet is filtered through the list from left to right until it meets a terminating action. Figure 3-9 details the processing of a filter list.

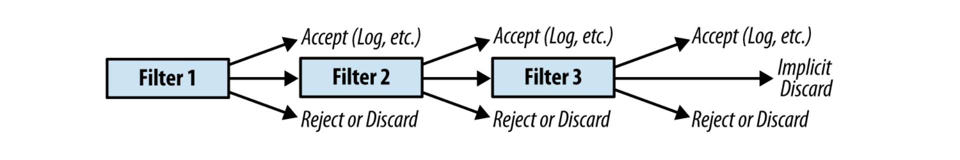


Figure 3-9. Filter List Processing.

The figure looks similar to Figure 3-1, which detailed term processing within a given filter, except now each block represents a *complete filter* as opposed to a specific term within a single filter. Each of the individual filters shown may consist of many individual terms. As with a single filter, there is still an implicit deny-all at the end of the filter list. While a single large filter can also perform the same functionality, many operators find it easier to manage several smaller filters that act in a modular fashion so they can be stacked together as needed in Lego fashion.

Filter Nesting

You can nest one filter into another using the `filter` keyword. A term that calls a nested filter cannot perform any matching or any other actions. Which is to say that if a firewall filter term includes the `filter` statement, it cannot include either a `from` or `then` condition. Nested filters are supported with standard stateless firewall filters only. You cannot use service filters or simple filters in a nested firewall filter configuration.

The total number of filters referenced from within a filter cannot exceed 256, and Junos currently supports a single level of filter nesting, which means you cannot do recursive filter nesting. For example, if `filter_1` references `filter_2`, then you cannot configure a `filter_3` that references `filter_1`, as this would require two levels of recursion. In like fashion, `filter_2` cannot reference `filter_1`, forming a loop.

Forwarding Table Filters

Figure 3-9 also shows filter application to the forwarding table, a function you configure at the `[edit forwarding-options]` hierarchy. Forwarding table filters are defined the same as other firewall filters, but you apply them differently. Instead of applying the filters to an interface, you apply them to a specific protocol families' Forwarding Table (FT) in either the input or output direction. An FT filter can act upon Broadcast, Unknown unicast, and Multicast (BUM) traffic, as a result of the FT lookup returning a flood-style next-hop, a function that's simply not possible with a filter or policer applied at the interface level.

FT filters are applied on a per-family basis, with most families supporting the input direction only; at this time, only the `inet` and `inet6` families support output direction FT filters as used to match on Destination Class Usage (DCU) information. The common use case in Layer 2-focused environments is to control the flow of BUM traffic for the `vpls` and `bridge` families, either through direct filtering action or by directing traffic to a policer for rate limiting.

In addition to matching on traffic types that require route/MAC lookup, for example to determine if a Unicast address has been learned or not, an FT filter can simplify filter administration by providing a single consolidated point for the control of traffic within a VLAN/bridge domain; because all traffic that is forwarded through a given instance consults that instance's FT, applying a single FT filter can be far more efficient then applying a given filter multiple times, once for each interface in the instance.

Keep these restrictions in mind when deploying bridge domain forwarding table filters:

- The filter cannot be interface specific.
- You can apply one filter and one flood filter only; filter lists are not supported.
- You cannot apply the same filter to both an interface and a bridge domain.

Though not the focus of this chapter, Figure 3-9 also shows the relationship of an FT filter applied under the `[edit forwarding-options]` hierarchy and the ones applied under the `[edit routing-options forwarding-table]` hierarchy, as the two filter types perform distinctly different roles that are often confused. The `routing-options` filter is used to control how the routing process installs routes into the FIB and can be applied as export only. The typical use case is per flow load balancing and/or fast recovery from forwarding path disruptions when using protection mechanisms like RSVP fast reroute. In both cases, you apply a per-packet load balancing policy to instruct the FIB to install (and possibly use) all next-hops for a given destination as opposed to just the currently selected one. Unlike simple load balancing, in the case of traffic protection the result is to install both the primary and backup next-hops into the FIB while only using the backup next-hop when the primary is unreachable.

General Filter Restrictions

Although you can use the same filter multiple times, you can apply only one input filter or one input filter list to an interface. The same is true for output filters and output filter lists.

Input and output filter lists are supported for the `ccc` and `mpls` protocol families except on management interfaces and internal Ethernet interfaces (`fxp0` or `em0`), loopback interfaces (`lo0`), and USB modem interfaces (`umd`).

On MX Series routers only, you cannot apply a Layer 2 CCC stateless firewall filter (a firewall filter configured at the `[edit firewall filter family ccc]` hierarchy level), as an output filter. On MX Series routers, firewall filters configured for the family `ccc` statement can be applied only as input filters.

Stateless filters configured at the `[edit firewall family any]` hierarchy, also known as protocol independent filters, are not supported on the router loopback interface (lo0).

Applying Policers

Policers can be applied directly to an interface or called via a filter to affect only matching traffic. In most cases, a policing function can be evoked at ingress, egress, or both. This means the same traffic can be subjected to several layers of policing, as shown in Figure 3-10.

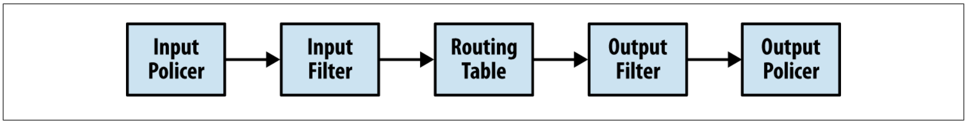


Figure 3-10. Policer Processing.

The figure shows how an interface policer is executed before any input filters and that the input filter can in turn call its own policer for matching traffic. The same is true for egress traffic, where it can be policed via a firewall-evoked policer before being subjected to the interface-level policer. In addition, for some families you can apply a policer via filter to the forwarding table, as needed to control BUM traffic in a Layer 2 switched environment.

Generally speaking, two-color and three-color policers can be applied in one of the following ways:

- As part of a filter in conjunction with MF classification at interface family level *http://www.juniper.net/techpubs/en_US/junos11.2/topics/concept/policer-types.html#jd0e197*
- Directly at interface family level as a logical interface policer
- Directly at interface logical unit level as a Layer 2 policer*http://www.juniper.net/techpubs/en_US/junos11.2/topics/concept/policer-types.html#jd0e192*
- Directly at interface family level as a physical interface policer

The characteristics and operation of all but the first policer application are detailed in the following. The topic of a firewall filter with a match condition with a policing action has been discussed.

Logical Interface Policers

A logical interface policer (also called an aggregate policer) is a policer that can police the traffic from multiple protocol families without requiring a separate instantiation of a policer for each such family on the IFL. You define a logical interface policer by including the `logical-interface-policer` statement when defining the policer.

There are two modes of operation for a logical interface policer; these are the protocol family and Layer 2 modes. The mode is determined on how you apply the policer to an interface.

In *protocol family mode*, you apply the logical interface policer under one or more protocol families that share a logical unit. While the policer will only affect the protocol families to which it has been applied, you can apply the same policer to different families on different IFL, with the result being a single instance of the policer that is shared by all associated protocol families under a given IFL. Stated differently, there is one policer instance for each IFL, regardless of how many families may live on that IFL.

The policer acts according to the layer associated with the family for which it's applied. For inet, that will be Layer 3 (i.e., the IP layer), whereas for the bridge family the policer functions at Layer 2, such that the policer includes frame overhead in its determination of in-profile traffic.

Currently, only two-color-style policers are supported in the protocol family mode.

A protocol family-based logical interface policer is shown:

```
regress@R1-RE0# show firewall policer family_police_mode
logical-interface-policer;
if-exceeding {
    bandwidth-limit 20k;
    burst-size-limit 2k;
}
then discard;
{master}[edit]
regress@R1-RE0# show interfaces ae0
flexible-vlan-tagging;
aggregated-ether-options {
    lacp {
        active;
        system-priority 100;
    }
}
unit 0 {
    family bridge {
        policer {
            input family_police_mode;
        }
        interface-mode trunk;
        vlan-id-list 1-999;
    }
}
unit 1 {
    vlan-id 1000;
    family inet {
        policer {
            input family_police_mode;
        }
        address 10.8.0.0/31;
    }
    family iso;
    family inet6 {
        policer {
            input family_police_mode;
        }
        address 2001::1/64;
    }
}
```

The example shows an AE interface with two IFLs, with IFL 1 having two Layer 3 protocol families while IFL 0 has a single Layer 2 family. Note how the *same* two-color policer is applied at the family level for both units, which means twice for IFL 1 given the two families. The net result is that three protocol families are policed while only two instances of the policer are required, one for each logical interface/IFL:

```
{master}[edit]
regress@R1-RE0# run show policer
Policers:
Name                                                Bytes             Packets
__default_arp_policer__                                 0                   0
family_police_mode-ae0.1-log_int-i                  18588                  39
family_police_mode -ae0.0-log_int-i                     0                   0
```

In Layer 2 mode, you apply the logical interface policer directly to a logical unit using the `layer2-policer` keyword. The policer now functions at Layer 2 for all families configured on that IFL. This differs from logical interface policer in protocol family mode, which is applied at a protocol family level such that its actions take effect at the layer associated with the family (i.e. at Layer 3 for `inet`). Further, you only have to apply this style policer once per unit, as opposed to the family mode where it's applied once per family when multiple families share a logical unit.

A Layer 2 policer counts at Layer 2, including Link Level encapsulation overhead and any CRC. In the case of untagged Ethernet there are 18 bytes of overhead per frame. Add four additional bytes for each VLAN tag.

Layer 2 mode logical interface policers support both single-rate two-color policer and three-color policer styles (either single-rate or two-rate). You must specify `color-blind` mode for a three-color policer when it operates as an input Layer 2 policer because input colors cannot be recognized at Layer 2. You can use color-aware for output Layer 2 policers, assuming a previous processing stage (such as a color-blind ingress Layer 2 policer) has colored the traffic.

As before, you need the `logical-interface-policer` statement in the policer definition, as this is still an example of a logical interface policer. Layer 2 mode is evoked based on the assignment of the policer at the logical unit rather than at the family level using the `layer2-policer` statement. You also configure the policer directionality and use of a two-color versus three-color policer type using the `input-policer`, `input-three-color`, `output-policer`, or `output-three-color` keywords.

Once applied, the Layer 2 policer operates at Layer 2, in the direction specified, for all protocol families that share the IFL. Note that you cannot apply a Layer 2 mode logical interface filter as a stateless firewall filter action. This differs from a protocol family-based logical interface policer, which can be applied directly at the family level or can be called as a result of filter action for the related family.

> Currently, both two-color and three-color policers are supported in Layer 2 mode, while only two-color policers are supported in protocol family mode unless a filter is used to evoke the logical interface policer.

A typical Layer 2 policer application is shown, in this case applied to the AE0 interface between R1 and R2:

```
{master}[edit firewall]
regress@R1-RE0# show
three-color-policer L2_mode_tcm {
    logical-interface-policer;
    action {
        loss-priority high then discard;
    }
    two-rate {
        color-blind;
        committed-information-rate 70k;
        committed-burst-size 1500;
        peak-information-rate 75k;
        peak-burst-size 2k;
    }
}

{master}[edit]
jnpr@R1-RE0# show interfaces ae0
flexible-vlan-tagging;
aggregated-ether-options {
    lacp {
        active;
        system-priority 100;
    }
}

unit 0 {
    layer2-policer {
        input-three-color L2_mode_tcm;
    }
    family bridge {
        interface-mode trunk;
        vlan-id-list 1-999;
    }
}

unit 1 {
    vlan-id 1000;
    layer2-policer {
        input-three-color L2_mode_tcm;
    }
    family inet {
        address 10.8.0.0/31;
    }
    family iso;
    family inet6 {
        address 2001::1/64;
    }
}
```

And once again, confirmation of a single policer instance (now at Layer 2) for each IFL is obtained with the CLI:

```
{master}[edit firewall]
regress@R1-RE0# run show interfaces policers ae0
Interface       Admin Link Proto Input Policer         Output Policer
ae0             up    up
ae0.0           up    up          L2_mode_tcm-ae0.0-ifl-i
                           bridge
Interface       Admin Link Proto Input Policer         Output Policer
ae0.1           up    up          L2_mode_tcm-ae0.1-ifl-i
                           inet
                           iso
                           inet6
                           multiservice __default_arp_policer__
Interface       Admin Link Proto Input Policer         Output Policer
ae0.32767       up    up
                           multiservice __default_arp_policer__
```

As additional confirmation, ping traffic is generated at R2 to test that the L2 policer is working for both the `inet` and `inet6` families:

```
{master}[edit]
jnpr@R2-RE0# run ping 2001::1 rapid count 100
PING6(56=40+8+8 bytes) 2001::2 --> 2001::1
!!!!!!!!!!!!!!!!!!!!.!!!!!!!!!!!!!!!!!!!!.!!!!!!!!!!!!!!!!!!!!.!!!!!!!!!!!!!!!!!!!!
--- 2001::1 ping6 statistics ---
100 packets transmitted, 96 packets received, 4% packet loss
round-trip min/avg/max/std-dev = 0.435/0.636/4.947/0.644 ms

{master}[edit]
jnpr@R2-RE0# run ping 10.8.0.0 rapid count 50
PING 10.8.0.0 (10.8.0.0): 56 data bytes
!!!!!!!!!!!!!!!.!!!!!!!!!!!!!!!.!!!!!!!!!!!!!!!.!!
--- 10.8.0.0 ping statistics ---
50 packets transmitted, 47 packets received, 6% packet loss
round-trip min/avg/max/stddev = 0.512/0.908/4.884/0.908 ms

{master}[edit]
jnpr@R2-RE0#
```

The ping loss for both families is a good sign that the policer is working. It is easy enough to verify with a `show interfaces` command for the logical interface in question at R1; note that Layer 2 policers are *not* shown in the output of the CLI's `show policers` command.

```
{master}[edit]
jnpr@R1-RE0# run show interfaces ae0.1 detail
  Logical interface ae0.1 (Index 324) (SNMP ifIndex 5534) (Generation 182)
    Flags: SNMP-Traps 0x4000 VLAN-Tag [ 0x8100.1000 ]  Encapsulation: ENET2
    Layer 2 input policer : L2_mode_tcm-ae0.1-ifl-i
    Loss Priority               Bytes                 Pkts                 Rate
    Low         :               9740                 112                   0 bps
    Medium-High :               2910                  33                   0 bps
    High        :                630                   7                   0 bps
    Dropped     :                630                   7                   0 bps
. . .
```

The output shows seven Layer 2 policer drops on R1's ae0.1 interface, which correlates exactly with the four lost IPv6 and the three lost IPv4 pings, confirming that the three-color policer marked a mix of `inet` and `inet6` packets as red with a corresponding drop action.

In both the protocol family and Layer 2 modes, a single policer can police traffic from multiple families on a per-IFL basis, hence the terms aggregate policer and logical interface policer are used interchangeably to describe its functionality.

Filter-Evoked Logical Interface Policers. Normally, you apply a logical interface policer directly to an interface, either at the logical unit or family level, depending on the mode desired, without referencing the policer through a filter. Starting with release v11.4, you can apply a logical interface policer on a Trio-based MX as an action in a firewall filter term, but you must also include the `interface-specific` statement in the calling filter in addition specifying to the `logical-interface-policer` statement in the related policer. Using a filter to evoke a logical interface filter has the added benefits of increased match flexibility as well as support for two-color policer styles, which can only be attached at the family level through a filter action.

Because the filter is applied at the family level, you cannot evoke a Layer 2 policer using the filter method. In addition, for each family you wish to be policed by the aggregate policer, you must apply a (family-specific) filter that evokes the common logical interface policer.

> By default, policers are term specific, which is to say that a separate policer instance is created when the same policer is referenced in multiple terms of a filter. Use the filter-specific keyword in the policer definition to later this behavior.

Physical Interface Policers

A physical interface policer is a two-color, three-color, or hierarchical policer that uses a single policer instance to rate limit all logical interfaces and protocol families configured on a physical interface, even if the logical interfaces belong to different routing instances or have mutually exclusive families such as `bridge` and `inet`. This feature enables you to use a single policer instance to perform aggregate policing for different protocol families and different logical interfaces on the same physical interface. Despite the name and function, you cannot apply a physical interface policer directly to a physical interface. Instead, you must evoke the common policer through multiple filter statements that are in turn attached to each protocol family on the various IFLs that share the physical interface.

You create a physical interface policer by including the `physical-interface-policer` statement in the policer definition. You must call this policer from a physical interface policer filter, which, as you may have guessed, is so designated when you add the `physical-interface-filter` statement to its definition. Again, note that you must create multiple such filter statements, because a separate (and uniquely named) family-specific filter is needed for each protocol family on the interface.

Please note the restriction and caveats that apply to a filter that references a physical interface policer:

- You must configure a specific firewall filter for each supported protocol family on the interface, which at this time are `inet`, `inet6`, `mpls`, `vpls`, or circuit cross-connect (`ccc`). You cannot define a physical interface filter under `family any`.
- You can not apply a physical interface policer directly to the interface (IFD) or logical units (IFL). You must call it at the protocol family level with a filter.
- You must designate the calling filter as a physical interface filter by including the `physical-interface-filter` statement within the filter configuration. You must also designate the policer by including the `physical-interface-policer` statement within the policer configuration.
- A firewall filter that is designated as a physical interface filter can only reference a policer that is also designated as a physical interface policer.
- A firewall filter that is defined as a physical interface filter cannot reference a policer configured with the `interface-specific` statement.
- You cannot configure a firewall filter as both a physical and logical interface filter.
- The single policer instance operates at Layer 2 if the first policer activated is for a Layer 2 family; otherwise, it's instantiated as a Layer 3 policer. This means Layer 3 traffic can be policed at Layer 2, or that Layer 2 traffic can be policed at Layer 3 depending on how the configuration is activated.

> You apply a physical interface policer to an interface that has a mix of Layer 2 and Layer 3 families. In such a case, the policer Layer 2 or Layer 3 mode is determined by the first family used to create the common policer. The result is that you may find you are policing Layer 2 traffic at Layer 3, or vice versa, both of which can lead to inaccuracies when compared to a pure Layer 2 or Layer 3 policer working on its respective protocol layers. To work around this behavior, you can commit the Layer 2 application first, and then apply to Layer 3 families, or use two different logical interface policers, one for all Layer 2 families and another for Layer 3.

A sample physical interface policer configuration is shown:

```
{master}[edit]
jnpr@R1-RE0# show firewall
```

```
family inet6 {
    filter inet6_phys_filter {
        physical-interface-filter;
        term 1 {
            then policer phys_policer;
        }
    }
}
family bridge {
    filter bridge_phys_filter {
        physical-interface-filter;
        term 1 {
            then policer phys_policer;
        }
    }
}
policer phys_policer {
    physical-interface-policer;
    if-exceeding {
        bandwidth-limit 50k;
        burst-size-limit 2k;
    }
    then discard;
}
filter inet_phys_filter {
    physical-interface-filter;
    term 1 {
        then policer phys_policer;
    }
}

{master}[edit]
jnpr@R1-RE0# show interfaces ae0
flexible-vlan-tagging;
aggregated-ether-options {
    lacp {
        active;
        system-priority 100;
    }
}
unit 0 {
    family bridge {
        filter {
            input bridge_phys_filter;
        }
        interface-mode trunk;
        vlan-id-list 1-999;
    }
}
unit 1 {
    vlan-id 1000;
    family inet {
        filter {
            input inet_phys_filter;
        }
        address 10.8.0.0/31;
    }
    family iso;
    family inet6 {
        filter {
            input inet6_phys_filter;
        }
        address 2001::1/64;
    }
}
```

In this example, a single policer named phys_policer will police the bridge, inet, and inet6 families on the ae0 interface. In this case, three filter statements are needed, one for each supported family, and each must be designated a physical interface filter. Also, the shared policer must include the physical-interface-policer statement. In this example, the goal is to match all traffic for aggregate policing; therefore, the filters use a single match all term and more complex filtering statements are possible.

Why No Filter/Policer for the iso Family?

The iso family is for support of IP routing using the IS-IS routing protocol. This family does not support filtering, nor should you attempt to subject it to a policer. It would be unusual to police your own network control plane as that could easily affect reconvergence when lots of protocol activity is expected. In addition, because IS-IS routing is transported directly in link-level frames, remote IS-IS exploits are very unlikely, which is not the case with OSPF and its IP level transport, especially given its domainwide flooding of external LSAs.

Policing remote access protocols is a different story. There are cases where you may want to police traffic that is allowed to flow to the control plane/routing engine, which is entirely different than simply filtering traffic that is not allowed to begin with. Routing engine protection from unwanted traffic, as well as from issues that stem from receiving too much of an allowed traffic type, is covered in the section on protecting the RE.

The filter application is confirmed:

```
{master}[edit]
jnpr@R1-RE0# run show interfaces filters ae0
Interface       Admin Link Proto Input Filter         Output Filter
ae0             up    up
ae0.0           up    up   bridge bridge_phys_filter-ae0-i
ae0.1           up    up   inet  inet_phys_filter-ae0-i
                           iso
                           inet6 inet6_phys_filter-ae0-i
                           multiservice
ae0.32767       up    up   multiservice
```

As expected, all three protocol families show their respective filters are applied. Extra tech credibility points for the reader that notes how here, unlike the previous logical interface policer example, the filter names are no longer associated with a unit. This is the result of the physical-interface-policer statement doing its job. Because the physical interface policer is called from a filter, it's not listed in the output of a show interfaces ae0 policers command (and so not shown, as there is therefore nothing interesting to see). Recall in the previous example on logical interface policers that there was no filter evocation, and it was confirmed that each of the two IFLs had a policer instance. Here, there is a single policer defined, and it's shared by all IFLs and supported families, as confirmed by looking at the policer itself on FPC 2, which houses the AE0 link members:

```
{master}[edit]
jnpr@R1-RE0# run request pfe execute target fpc2 command "show filter shared-pol"
SENT: Ukern command: show filter shared-pol
GOT:
GOT: Policers
GOT: --------
GOT: Name                                    Location    RefCount    Size
GOT:              phys_policer-filter-ae0-i  0          3        0
GOT:
GOT: Tricolor Policers
GOT: -----------------
GOT: Name                                    Location    RefCount    Size
LOCAL: End of file
```

If desired, you can display the filter and its related policer properties. Here, it is confirmed to be an IFD-level policer by virtue of the fact that while three filters exist only one policer instance is found:

```
{master}[edit]
jnpr@R1-RE0# run request pfe execute target fpc2 command "show filter"
SENT: Ukern command: show filter
GOT:
GOT: Program Filters:
GOT: ---------------
GOT:    Index     Dir     Cnt    Text     Bss  Name
GOT: --------  ------  ------  ------  ------  --------
GOT:
GOT: Term Filters:
GOT: ------------
GOT:    Index    Semantic   Name
GOT: --------  ---------- ------
GOT:        2  Classic    __default_bpdu_filter__
GOT:        5  Classic    inet_phys_filter-ae0-i
GOT:        6  Classic    inet6_phys_filter-ae0-i
GOT:        7  Classic    bridge_phys_filter-ae0-i
GOT:    17000  Classic    __default_arp_policer__
. . .
LOCAL: End of file

{master}[edit]
jnpr@R1-RE0# run request pfe execute target fpc2 command "show filter index 5 policers"
SENT: Ukern command: show filter index 5 policers
GOT:
GOT: Instance name                        Bw limit-bits/sec  Burst-bytes      Scope
GOT: -------------                        -----------------  -----------      -----
```

```
GOT: phys_policer-filter                  50000              2000             ifd
LOCAL: End of file
```

And to test that all is indeed working traffic is generated from R2:

```
{master}[edit]
jnpr@R2-RE0# run ping 2001::1 rapid count 10 size 1000
PING6(1048=40+8+1000 bytes) 2001::2 --> 2001::1
!.!.!.!.!.
--- 2001::1 ping6 statistics ---
10 packets transmitted, 5 packets received, 50% packet loss
round-trip min/avg/max/std-dev = 0.819/5.495/23.137/8.827 ms
{master}[edit]

jnpr@R2-RE0# run ping 10.8.0.0 rapid count 10 size 1000
PING 10.8.0.0 (10.8.0.0): 1000 data bytes
!.!.!.!.!.
--- 10.8.0.0 ping statistics ---
10 packets transmitted, 5 packets received, 50% packet loss
round-trip min/avg/max/stddev = 0.796/1.553/4.531/1.489 ms
```

The policer count is displayed to confirm equal actions are reported against all three families, again evidence of a single policer instantiation:

```
{master}[edit]
jnpr@R1-RE0# run show firewall

Filter: __default_bpdu_filter__

Filter: inet_phys_filter-ae0-i
Policers:
Name                                                Bytes              Packets
phys_policer-filter-ae0-i                           10380                   10

Filter: inet6_phys_filter-ae0-i
Policers:
Name                                                Bytes              Packets
phys_policer-filter-ae0-i                           10380                   10

Filter: bridge_phys_filter-ae0-i
Policers:
Name                                                Bytes              Packets
phys_policer-filter-ae0-i                           10380                   10
```

It's worth noting again that the combination of Layer 2 and Layer 3 families in this example results in a shared policer that must operate at *either* Layer 2 or Layer 3 for all families, with the policer type a function of which family instantiates it first. In this example, it was found that `family bridge` created the policer, so it operates at Layer 2 for all families. This is confirmed by clearing the counters and generating traffic with a single IPv4 ping using 100 bytes of payload:

```
{master}
jnpr@R2-RE0> ping 10.8.0.0 rapid count 50 size 100
PING 10.8.0.0 (10.8.0.0): 100 data bytes
!!!!!!!!!!!!.!!!!!!!!!!!!.!!!!!!!!!!!!!.!!!!!!!
```

```
--- 10.8.0.0 ping statistics ---
50 packets transmitted, 47 packets received, 6% packet loss
round-trip min/avg/max/stddev = 0.519/1.661/27.622/4.176 ms
```

With 100 bytes of payload and 20 + 8 bytes of IP/ICMP header, respectively, a Layer 3 policer should have counted 128 bytes per Layer 3 datagram. Looking at packet versus byte counters on R1's policer, it is evident that each packet was in fact 22 bytes longer, showing 150 bytes per policed packet:

```
. . .
Filter: bridge_phys_filter-ae0-i
Policers:
Name                                                    Bytes              Packets
phys_policer-filter-ae0-i                                 450                    3
```

The extra 22 bytes seen in the output are the Ethernet MAC addressing, type code, and 4-byte FCS. Though not shown, the author removed the `family bridge` filter application on ae0.0 and reactivated the filter configuration with only Layer 3 families present to ensure a Layer 3 policer is created. The family bridge filter is then added back and a L3 policer is confirmed:

```
{master}[edit]
jnpr@R1-RE0# run show firewall

Filter: inet_phys_filter-ae0-i
Policers:
Name                                                    Bytes              Packets
phys_policer-filter-ae0-i                                 384                    3

Filter: __default_bpdu_filter__

Filter: inet6_phys_filter-ae0-i
Policers:
Name                                                    Bytes              Packets
phys_policer-filter-ae0-i                                 384                    3

Filter: bridge_phys_filter-ae0-i
Policers:
Name                                                    Bytes              Packets
phys_policer-filter-ae0-i                                 384                    3
```

Now, the same IP traffic results in three discards for a total of 384 octets, which yields the expected 128 bytes per packet, and thereby confirms Layer 3 policer operation.

Policer Application Restrictions

The following general guidelines should be kept in mind when deploying Junos policers:

- Only one type of policer can be applied to the input or output of the same physical or logical interface. For example, you are not allowed to apply a basic two-color policer and a hierarchical policer in the same direction at the same logical interface.
- You cannot chain policers, which is to say that applying policers to both a port and to a logical interface of that port is not allowed.

Bridge Filtering Case Study

Up to this point, we have been discussing general MX platform filtering and policing capabilities. This section focuses on a practical filtering and policing deployment, along with the operational mode commands used to validate and confirm filter and policer operation in Junos.

Filter Processing in Bridged and Routed Environments

Before jumping into the case study, a brief review of packet flow and filter processing for MX routers that support simultaneous Layer 2 bridging and Layer 3 routing is in order. This is not only a common use case for the MX, but is also the basics for the upcoming case study, so be sure to follow along.

MX routers use an Integrated Routing Bridging (IRB) interface to interconnect a Bridge Domain (BD) with its Layer 3 routing functionality. On the MX, when traffic arrives at an L2 interface, it's first inspected to determine whether it needs bridging, routing, or both.

Routed

If the packet's L2 destination MAC address matches the router's IRB MAC address, the traffic is routed. In this case, the traffic is mapped to the BD's IRB, and all filters (or route table lookup) are driven by the IRB configuration. It's important to note that any bridge family filters applied to the related Layer 2 IFLs, or to the FT in the BD itself, are not evaluated or processed for routed traffic, even though that traffic may ingress on a Layer 2 interface where a Layer 2 input filter is applied.

Bridged

When a packet's destination MAC address (DMAC) is unicast but *does not match* the IRB MAC address, that traffic is bridged. Bridged traffic is mapped to the L2 IFLs, and any input or output Layer 2 bridge filters are evaluated. In addition, if the DMAC is also unknown (making this unknown unicast), any BD-level BUM filters are also evaluated.

Brouted (bridged and routed)

Given that it's necessary for a switching and routing book to combine the terms bridged and routed into *brouted*, at least once, we can say that obligation has been met and move on to bigger things.

> When a packet's DA is broadcast or multicast, the traffic type is IPv4, IPv6, or ARP, and the BD has an IRB configured, then the packet is copied. The original packet is given the bridge treatment while the copy is afforded the IRB/L3 treatment. In such cases, the packets can be flooded in the BD while also being routed, as is common in the case of multicast IGMP queries, for example.
>
> In such cases, it should be noted that Layer 3 filters (applied at an IRB) cannot match based on L2 MAC address or the EtherType because these fields are stripped once the decision is made to route the packet rather than to bridge. However, a Layer 2 filter that is applied to bridged traffic is able to match on any of the supported Layer 2, Layer 3, or Layer 4 fields, given no information is stripped from bridged traffic.

An interesting side effect of this is that you have to place an *output* filter on a bridge domain's IRB when you wish to match on or filter locally generated Layer 3 traffic (such as a telnet session from the RE) that is injected into a Layer 2 bridge domain.

Monitor and Troubleshoot Filters and Policers

There are a several commands and techniques that are useful when verifying filter and policer operation, or when attempting to diagnose the lack thereof. Most have already been shown in the preceding discussion, but here the focus is on operational verification and troubleshooting of Junos filters and policers.

The most common operational mode commands for filters include the `show firewall`, `clear firewall`, and `show interface filters` commands. For interface policers, you have the `show policer` and `show interface policers` commands. As for useful techniques, the list includes monitoring the syslog log for any errors at time of commit, or adding log or count terms to a filter to make debugging the matching of various types of traffic easier.

Note that currently there is no clear policer command. To clear an interface policer count (for a policer applied directly to an interface and not via a firewall), use the `clear firewall all` command. This command clears all filter and policer counts—there is no way to specify just the interface policer you wish to clear, but deactivating the policer application and then restoring should be a workaround when you do not wish to clear all counters.

To help illustrate how each command is used, consider this simple RE filter application:

```
{master}[edit]
jnpr@R1-RE0# show firewall filter re_filter
interface-specific;
term 1 {
    from {
        protocol icmp;
    }
    then {
        policer icmp_police;
        count icmp_counter;
    }
}
term 2 {
    then {
        count other;
        accept;
    }
}
{master}[edit]
jnpr@R1-RE0# show firewall policer icmp_police
if-exceeding {
    bandwidth-limit 20k;
    burst-size-limit 2k;
}
then forwarding-class fc0;
{master}[edit]
jnpr@R1-RE0# show interfaces lo0
unit 0 {
    family inet {
        filter {
            input re_filter;
        }
        address 10.3.255.1/32;
    }
    family iso {
        address 49.0001.0100.0325.5001.00;
    }
}
```

It's noted that in the current test bed, R1 has no protocols configured and the CLI is accessed via the console port so as to not generate any fxp0/lo0 traffic. You therefore expect the RE's lo0 interface to be quiet unless stimulated with some ICMP test traffic.

Things start with confirmation that the filter exists and is applied to the lo0 interface in the input direction:

```
{master}[edit]
jnpr@R1-RE0# run show interfaces filters lo0
Interface       Admin Link Proto Input Filter         Output Filter
lo0             up    up
lo0.0           up    up   inet  re_filter-lo0.0-i
                           iso
lo0.16384       up    up   inet
lo0.16385       up    up   inet
```

Satisfied the filter is well and truly applied, issue a `show firewall` command, which is, of course, one of the primary methods used to monitor filter operation. With no arguments added, the output lists all filters and their filter-evoked policers, along with any counter values associated with the `count` action modifier in those filters. You can specify a filter name to view just that individual filter's statistics, and when the filter has many terms with counters, you can also add the counter name to reduce clutter.

```
{master}[edit]
jnpr@R1-RE0# run show firewall

Filter: __default_bpdu_filter__

Filter: re_filter-lo0.0-i
Counters:
Name                                                Bytes              Packets
icmp_counter-lo0.0-i                                    0                    0
other-lo0.0-i                                           0                    0
Policers:
Name                                                Bytes              Packets
icmp_police-1-lo0.0-i                                   0                    0
```

The output confirms a filter named `re_filter` is defined, and that the filter has two counters, one named `icmp_counter` and the other, which here is given the rather uncreative name of `other`. The display confirms that the filter is bound to a policer named `icmp_police`, and that neither counter is cranking, which is in keeping with the belief that the system's lo0 interface is quiescent. The policer count of 0 confirms that no out-of-profile ICMP traffic has been detected, which is expected, given the `icmp_counter-lo0.0-i` term counter indicates there is no ICMP traffic.

Note that the filter and counter names have been appended with information indicating the applied interface, lo0, and the directionality of the filter, which in this case uses `i` for input. Recall that the system-generated name of an `interface-specific` firewall filter counter consists of the name of the configured counter followed by a hyphen ('`-`'), the full interface name, and either '`-i`' for an input filter instance or '`-o`' for an output filter instance. These name extensions are automatically added when a filter is made interface specific as each application of that same filter requires a unique set of counters.

The display also confirms an automatically created filter called `__default_bpdu_filter__`. This filter is placed into effect for VPLS routing instances to perform multifield classification on STP BPDUS (by matching on the well-known STP multicast destination MAC address of 01:80:C2:00:00:00), so they can be classified as Network Control and placed into queue 3. You can override the default BPDU filter with one of your own design by applying yours as a forwarding table filter in the desired VPLS instance.

The command is modified to display only the counter associated with the policer named `icmp_policer`:

```
jnpr@R1-RE0# run show firewall filter re_filter-lo0.0-i counter icmp_police-1-lo0.0-i
Filter: re_filter-lo0.0-i
Policers:
Name                                                Bytes              Packets
icmp_police-1-lo0.0-i                                   0                    0
```

Things look as expected thus far, so ICMP traffic is generated from R2; note that while there are no routing protocols running, there is a direct link with a shared subnet that allows the ping to be routed.

```
{master}
jnpr@R2-RE0> ping 10.8.0.0 count 4
PING 10.8.0.0 (10.8.0.0): 56 data bytes
64 bytes from 10.8.0.0: icmp_seq=0 ttl=64 time=0.691 ms
64 bytes from 10.8.0.0: icmp_seq=1 ttl=64 time=0.683 ms
64 bytes from 10.8.0.0: icmp_seq=2 ttl=64 time=0.520 ms
64 bytes from 10.8.0.0: icmp_seq=3 ttl=64 time=0.661 ms

--- 10.8.0.0 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.520/0.639/0.691/0.069 ms
```

The pings succeed, which is a most auspicious sign; confirm the count function in the `icmp_counter-lo0` term:

```
{master}[edit]
jnpr@R1-RE0# run show firewall counter icmp_counter-lo0.0-i filter re_filter-lo0.0-i
Filter: re_filter-lo0.0-i
Counters:
Name                                                Bytes              Packets
icmp_counter-lo0.0-i                                  336                    4
```

The counter reflects the four ping packets sent and correctly tallies their cumulative byte count at the IP layer, which makes sense as this is an `inet` family policer and it therefore never sees any Layer 2. Here four IP packets were sent, each with a payload of 64 bytes, of which 8 are the ICMP header. Each packet has a 20 byte IP header, making the total 4 * (64 + 20), which comes out nicely to 336 bytes.

The ICMP policer is again displayed to verify that none of the (paltry) pings exceeded the policer's bandwidth or burst settings, as indicated by the ongoing 0 count.

```
{master}[edit]
jnpr@R1-RE0# run show firewall filter re_filter-lo0.0-i counter icmp_police-1-lo0.0-i

Filter: re_filter-lo0.0-i
Policers:
Name                                                Bytes              Packets
icmp_police-1-lo0.0-i                                   0                    0
```

Recall that in this example the policer action is set to alter the traffic's forwarding class so you cannot expect ping failures for each case of out-of-profile ICMP traffic. For the ping to actually fail the packet, now in Best-Effort fc0, it would have to meet with congestion and suffer a WRED discard. For now, you can assume there is no link congestion in the MX lab so no loss is expected for out-of-profile traffic.

The firewall filter and related policer counts are cleared at R1 for a fresh start:

```
{master}[edit]
jnpr@R1-RE0# run clear firewall all
```

Back at r2, increase the rate of ping generated while also increasing the packet size to 1,000 bytes. The goal is to generate four large pings in a very short period of time:

```
{master}
jnpr@R2-RE0> ping 10.8.0.0 count 4 rapid size 1000
PING 10.8.0.0 (10.8.0.0): 1000 data bytes
!!!!
--- 10.8.0.0 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.712/19.354/41.876/18.011 ms
```

The filter statistics are again displayed:

```
{master}[edit]
jnpr@R1-RE0# run show firewall filter re_filter-lo0.0-i

Filter: re_filter-lo0.0-i
Counters:
Name                                                Bytes              Packets
icmp_counter-lo0.0-i                                 4112                    4
other-lo0.0-i                                          76                    1
Policers:
Name                                                Bytes              Packets
icmp_police-1-lo0.0-i                                6168                    6
```

The output confirms two things; first, the rapid pings at R2 exceeded the ICMP policer's profile, as evidenced by its non-zero packet and byte count, and secondly, that the lo0 interface is not as quiet as you imagined, given the `other` term now shows a non-zero count.

> The discrepancy between the ICMP and policer counters was noted, and PR 719192 was raised to track the issue.

Hmm, what can that be? Getting a quick answer is a classic use case of the `log` (or `syslog`) action modifiers. Here, the former is used as no syslog changes are needed, and a permanent record of the traffic is not yet desired. In this case, you just want a quick and easy answer as to what traffic is flowing to the RE via lo0, in the absence of your pings. The filter is modified to include the `log` action modifier in term `2`:

```
{master}[edit]
jnpr@R1-RE0# edit firewall filter re_filter

{master}[edit firewall filter re_filter]
jnpr@R1-RE0# set term 2 then log

{master}[edit firewall filter re_filter]
jnpr@R1-RE0# commit
. . .
```

After a few moments, display the filter log with a `show firewall log` command.

```
{master}[edit firewall filter re_filter]
jnpr@R1-RE0# run show firewall log
Log :
Time      Filter    Action Interface     Protocol    Src Addr         Dest Addr
17:34:41  re_filter-lo0.0-i A fxp0.0     UDP         172.19.91.43     172.19.91.255
17:34:39  re_filter-lo0.0-i A fxp0.0     UDP         172.17.27.46     172.19.90.172
17:34:38  re_filter-lo0.0-i A fxp0.0     IGMP        172.19.91.95     224.0.0.1
16:06:19  pfe       A      unknown       VRRP        192.0.2.67       224.0.0.18
. . .
```

The output is a relative gold mine of information. Following the action is a bit easier if you note that R1's current fxp0 address is 172.19.90.172/23.

The basic display lists the timestamp, protocol, filter name, filter action, interface on which the packet arrived (or exited for an output filter), and both the source and destination IP addresses. Starting at the last entry, note that the VRRP traffic does not display a filter name—instead, it simply lists `pfe` in the filter column. This is expected, because the PFE is a data plane entity and does not store the user-assigned names of filters. This tells you that the VRRP traffic encountered a copy of the `re_filter-lo0.0-I` filter in the PFE. Stated differently, all the logged traffic arrived on the fxp0 interface and encountered the RE copy of the filter (and the RE knows the filter name), with the exception of the VRRP traffic.

The next entry is multicast-related IGMP that is sent to the all multicast host's well-known group address 224.0.0.1. Its presence tells you something is running multicast on the OoB management network, and that R1 considers itself a multicast host (though this does not imply it is running a multicast routing protocol, like PIM) in that its NIC has been told to accept traffic sent to the associated multicast MAC address; only traffic sent to the NIC's unicast address or one of its multicast groups will be accepted for further process, and thus make it far enough to hit the filter.

The next entry indicates that a UDP packet was sent by 172.17.27.43 to R1's fxp0 address, where it was received, naturally enough, on the fxp0 interface. Exactly what was in that UDP datagram is anyone's guess given no ports are shown in the basic display. Growing concerned? You may be the victim of some nefarious individual who's attempting to disrupt the router's operation; you add the detail switch to view the related protocol ports (when applicable):

```
jnpr@R1-RE0# run show firewall log detail
. . .
Name of protocol: UDP, Packet Length: 0, Source address: 172.19.91.46:138,
  Destination address: 172.19.90.172:138
Time of Log: 2011-12-06 17:34:39 PST, Filter: re_filter-lo0.0-i,
  Filter action: accept, Name of interface: fxp0.0
```

The detailed output shows the UDP packet was sent from and to port 138, the assigned port for the NetBIOS datagram service. From this, you conclude there are some Windows machines (or at least SMB servers) present on the OoB management network. Given that NetBIOS is a "chatty" protocol that likes to do name and service discover, it can be assumed this is the cost of having Windows services on the management network; the mere presence of this packet simply means it was sent to R1. It does not in itself confirm that R1 is actually listening to that port, nor that it's running any kind of Windows-based file or print services (and it's not).

The last entry is also some type of UDP packet, this time sent to destination IP address 172.19.91.255. At first, this may seem surprising, given that is not the address assigned to r1's fxp0. Looking back at its IP address, you realize this is the directed subnet broadcast address for the fxp0 interface's 172.19.90/23 subnet. The use of broadcast again explains why R1 has chosen to receive the traffic, and again does not imply it actually cares. All hosts receive directed IP subnet broadcast, but as they process the message may silently discard the traffic if there is no listing process, which is again the case here.

Satisfied the filter is working as expected, the log action is removed to reduce filter resource usage. Removing any counters or log action modifiers that are no longer needed is a best practice for Junos firewall filters.

Monitor System Log for Errors

Junos supports rich system logging; in many cases, error messages reporting incompatible hardware or general misconfigurations that are simply not reported to the operator at the time of commit can be found in the syslog. Many hours of otherwise unproductive troubleshooting can be saved with one error message. For example, when developing this material, a trTCM policer was applied at Layer 2 to an aggregated Ethernet interface. The policer was configured with a low CIR of 20 Kbps to ensure that it would catch ping traffic; the new policer and its application to an AE interface *appeared* to commit with no problems. However, no policer was found; a quick look at the `messages` file around the time of the commit did much to demystify the situation:

```
Dec  7 16:25:28  R1-RE0 dcd[1534]: ae2 : aggregated-ether-options link-speed set to
  kernel value of  10000000000
Dec  7 16:25:28  R1-RE0 dfwd[1538]: UI_CONFIGURATION_ERROR: Process: dfwd, path:
  [edit interfaces ae0 unit 0], statement: layer2-policer,  Failure to add
  Layer 2 three-color policer
Dec  7 16:25:28  R1-RE0 dfwd[1538]: Configured bandwidth 20000 for layer2-policer
  L2_mode_tcm is less than minimum supported bandwidth 65536
Dec  7 16:25:28  R1-RE0 dfwd[1538]: UI_CONFIGURATION_ERROR: Process: dfwd, path:
  [edit interfaces ae0 unit 1], statement: layer2-policer,  Failure to add
  Layer 2 three-color policer
Dec  7 16:25:28  R1-RE0 dfwd[1538]: Configured bandwidth 20000 for layer2-policer
  L2_mode_tcm is less than minimum supported bandwidth 65536
```

Adjusting the bandwidth to the minimum specific resolved the issue, and the policer was correctly created.

Bridge Family Filter and Policing Case Study

In this section, you deploy stateless filters and policing for the bridge family and confirm all operating requirement are met using CLI operational mode commands. Refer to Figure 3-11 for the topology details.

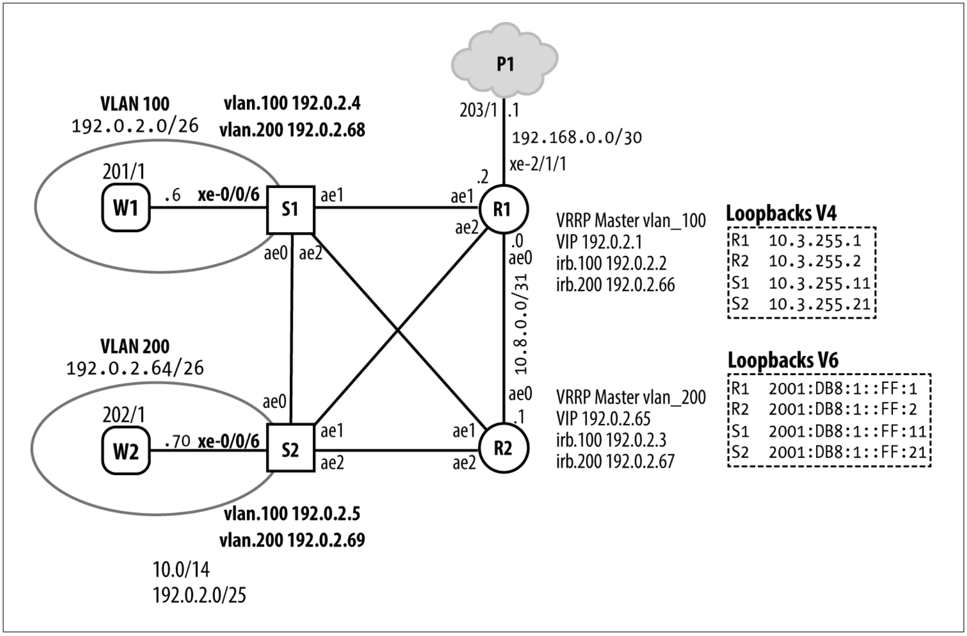


Figure 3-11. Bridge Filtering Topology.

The Layer 2/Layer 3 hybrid topology is based on the standard topology discussed in the Preface. The main areas to note are the two VLANs, `vlan_100` and `vlan_200`, along with their assigned logical IP subnets (LIS). The AE link between R1 and R2 has two IFLs: one is provisioned for Layer 2 via the bridge family and the other for Layer 3 via the `inet` and `inet6` families. IS-IS is running on the Layer 3 units between R1 and R2 to advertise lo0 routes. In addition, note that two VRRP groups are configured on the IRB interfaces at R1 and R2, with R1 the VIP master for `vlan_100` and vice versa. Two `vlan` interfaces are defined on each EX switch with an IP address in both VLANs to facilitate test traffic generation. VTSP is provided such that R1 is the root of `vlan_100` while R2 heads up `vlan_200`.

To complete the case study, you must alter the configuration of the network to meet the following requirements:

Ensure that VLAN 100 flooding cannot exceed:
Broadcast 1 Mbps/2 msec burst
Unknown Unicast 250 kbps/2 msec burst

Multicast 4 Mbps/4 msec burst

Prevent (and count) outgoing IPv4 HTTP connection requests from leaving VLAN 100

Given the criteria, it's clear you need to use a filter to match on and block HTTP connection requests while also directing BUM traffic to a suitable set of policers. Though not stated, the need to police intra-VLAN communications forces you to apply your changes to both R1 and R2 to cover the case of R2 being the active root bridge for VLAN 100 should R1 suffer some misfortune.

There is no need for three-color marking, and the specified sustained and allowed bursts rates can be accommodated with a classical two-color policer. Note that a logical interface policer does not work here as it would catch all the Layer 2 traffic, not just BUM, and besides, interface policers (direct or filter-evoked) can only be used for unicast traffic anyway. Yet, the need to selectively police BUM traffic in a Layer 2 domain necessitates the use of one or more family bridge filters to accomplish your task. Thus, the question becomes "Where should you apply these filters?"

The answer is twofold, and it may help to refer back to Figure 3-8 on filter and policer application points before considering your answer final.

Tackling the BUM issue first, you need to apply the filter to the FT using the `forwarding-options` hierarchy to filter and/or police BUM traffic. As stated previously, the determination of traffic as being type BUM is only possible after the MAC address has been looked up in the FT, hence an interface input filter does not work. Given that Unknown Unicast type match is not supported for output filters in Trio, the use of output filters on the egress interfaces is also off the table.

The second requirement needs to be examined closely. If the goal was to block HTTP request *within* VLAN 100, then you could opt for a FT filter (preferred), or if work is your bag you can apply a family bridge filter as input (or perhaps output if you do not mind doing a MAC lookup only to then discard) to all interfaces associated with trunking VLAN 100. Here, the goal is to prevent HTTP requests from *leaving* the VLAN, making this is an inter-VLAN filtering application; therefore, interface-level filters within VLAN 100 do not meet the requirements. Thinking back on inter-VLAN routing, you recall that the IRB interface functions as the default gateway for inter-VLAN communications (routing). Therefore, applying the filter to the IRB interfaces in VLAN 100 should accomplish the stated objective.

Policer Definition

With a plan in place, things begin with definition of the three policers to be used collectivity to limit BUM traffic with VLAN 100. Note the discard action, as needed to ensure traffic cannot exceed the specified limits, whether or not congestion is present within the network:

```
policer vlan100_broadcast {
    if-exceeding {
        bandwidth-limit 1m;
        burst-size-limit 50k;
    }
    then discard;
}


policer vlan100_unknown_unicast {
    if-exceeding {
        bandwidth-limit 250k;
        burst-size-limit 50k;
    }
    then discard;
}


policer vlan100_multicast {
    if-exceeding {
        bandwidth-limit 4m;
        burst-size-limit 100k;
    }
    then discard;
}
```

The policer burst rates are in bytes and based on each aggregated Ethernet interface in the lab having two 100 Mbps members. Not worrying about the interframe gap, such an AE interface sends some 200 Kbps each millisecond, which divided by 8 yields 25K bytes per millisecond. The 25 Kbps value was then used for the two specified burst tolerances, yielding the 50 Kbps and 100 Kbps values for the 2 and 4 millisecond criteria, respectively.

HTTP Filter Definition

The HTTP filter is now defined. Note the that the filter is defined under the family `inet`, given it will be applied to a Layer 3 interface (the IRB), and how the match criteria begin at Layer 4 by specifying a TCP protocol match along with a destination port match of either 80 or 443; though not specifically stated, both the HTTP and secure HTTP ports are specified to block both plain text and encrypted HTTP connection requests from being sent to servers on these well-known ports.

```
{master}[edit]
jnpr@R1-RE0# show firewall filter discard-vlan100-http-initial
term 1 {
    from {
        protocol tcp;
        destination-port [ http https ];
        tcp-initial;
    }
    then count discard_vlan100_http_initial;
}
term 2 {
    then accept;
}
```

Because only initial connection requests are to be matched, the filter must also look for the specific TCP flag settings that indicate the first segment sent as part of TCP connection establishment. These initial segments have a set SYN bit and a reset (or cleared) ACK bit. For `inet` and `inet6` family filters, this common TCP flag combination has been assigned an easy to interpret text synonym of `tcp-initial`, as used in the sample filter. The same effects are possible with the `tcp-flags` keyword along with text alias or hexadecimal-based specification of flag values. When manually specifying TCP flag values you use a not character (`!`) to indicate a reset (`0`) condition, or the flag is assumed to be set for a match to occur.

TCP Flag Matching for Family Bridge

It's pretty impressive that a Trio PFE can function in Layer 2 mode and yet still be capable of performing pattern matching against Layer 4 transport protocols. The `inet` filter example shown in this section uses the `tcp-initial keyword`. Unlike the Layer 3 families, the `bridge` family currently requires the use of the `tcp-flags` keyword along with either a text alias or hexadecimal-based entry for the desired flag settings. For a `family bridge` filter, you can match the effects of `tcp-initial` using the `tcp-flags` keyword, as shown in the following. Note how the filters specify the same ending match point, but here begins at Layer 2 with an EtherType match for the IPv4 protocol:

```
{master}[edit firewall]
jnpr@R1-RE0# show family bridge
filter discard-vlan100-http-initial {
    term 1 {
        from {
            vlan-ether-type ipv4;
            ip-protocol tcp;
            destination-port [ http https ];
            tcp-flags "(syn & !ack)";
        }
        then count discard_vlan100_http_initial;
    }
    term 2 {
        then accept;
    }
}
```

In similar fashion, specifying `tcp-flags "(ack | rst)"` mimics the functionality of an `inet` filter's `tcp-established` keyword, which is used to match on all TCP segments except the initial one.

As noted in the planning phase, the filter is applied to the IRB interface serving VLAN 100, at both R1 and R2, under the `inet` family; however, only the changes made at R1 are shown for brevity. A similar `inet6` filter could be applied if blocking IPv6-based HTTP was also a requirement. Applying the filter in the *input* direction is a critical part of achieving the desired behavior of blocking outgoing, rather than incoming requests. While using input filter to block traffic from going out of the VLAN may not seem intuitive, it makes sense, perhaps more so after looking at Figure 3-12.

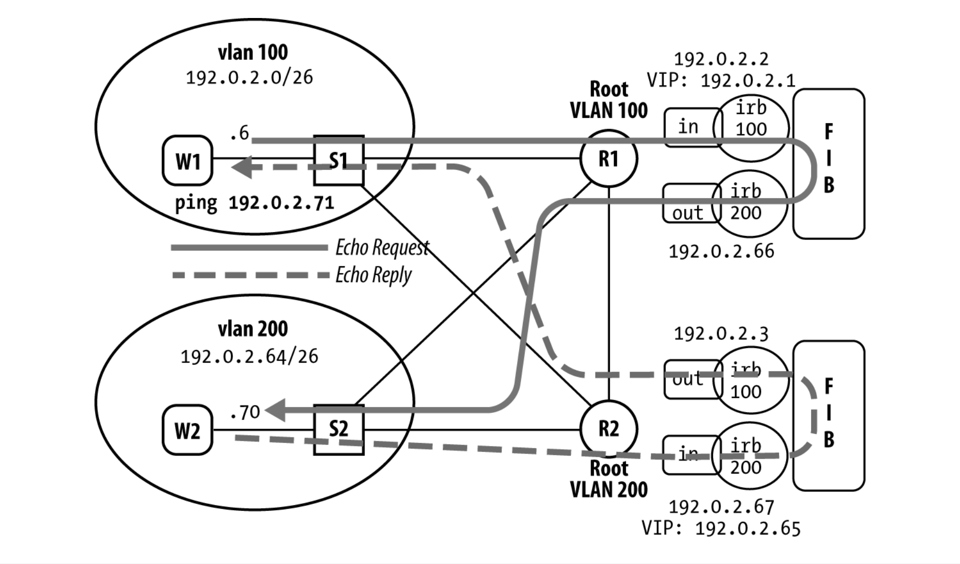


Figure 3-12. Inter-VLAN communications and IRB Filter Directionality.

The figure shows the flow of inter-VLAN communications through the two IRB interfaces for traffic between hosts in separate VLANs. It's clear that traffic leaving a VLAN flows *into* the VLAN's IRB, and then on to the next-hop destination based on a routing table lookup. Having arrived from a VLAN, the now routed traffic could very well egress on a core-facing 10 GE interface, which of course has precious little to do with any output filter you might have applied to the VLAN's IRB. Though a bit counterintuitive, applying a filter in the *output* direction of a VLAN IRB affects traffic that might *arrive* from a core interface and then be routed *into* the VLAN, which means the traffic does in fact *exit* on the IRB, and is thus affected by an *output* filter. As the goal is to block traffic arriving from VLAN 100 as it exits the VLAN, an input filter is required in this case:

```
{master}[edit]
jnpr@R1-RE0# show interfaces irb.100
family inet {
    filter {
        input discard_vlan100_http_initial;
    }
    address 192.0.2.2/26 {
        vrrp-group 0 {
            virtual-address 192.0.2.1;
            priority 101;
            preempt;
            accept-data;
        }
    }
}
```

Flood Filter

The flood filter is now created and applied. Note this filter is defined under `family bridge`, which is required given its future use for a Layer 2 application to a bridging domain.

```
{master}[edit]
jnpr@R1-RE0# show firewall family bridge
filter vlan_100_BUM_flood {
    term police_unicast_flood {
        from {
            traffic-type unknown-unicast;
        }
        then {
            policer vlan100_unknown_unicast;
            count vlan100_unicast_flood_allowed;
        }
    }
    term broadcast_flood {
        from {
            traffic-type broadcast;
        }
        then {
            policer vlan100_broadcast;
            count vlan100_bcast_flood_allowed;
        }
    }
    term mcast_flood {
        from {
            traffic-type multicast;
        }
        then {
            policer vlan100_multicast;
            count vlan100_mcast_flood_allowed;
        }
    }
}
```

The filter terms, one each for Broadcast, Unknown Unicast, and Multicast (BUM), all match on their respective traffic type and direct matching traffic to both a policer and a count action. As per the counter names, the goal is to count how much of each traffic type was flooded, as well as being able to use the policer counters to determine how much was blocked due to being out of profile in the event that adjustments are needed to support valid flooding levels.

Given the traffic type-based match condition, the flood filter cannot be applied at the interface level given that unknown unicast is not supported as an input match condition and broadcast is not supported as an output match condition. Even if such match types were supported, as the goal is to affect all of VLAN 100's flood traffic, applying such a filter to a large number of trunk or access interfaces would quickly prove a burden. By applying as a flood filter to a bridge domain's forwarding table, you affect all traffic within that domain, which fits the bill nicely here.

```
{master}[edit]
jnpr@R1-RE0# show bridge-domains VLAN100
vlan-id 100;
routing-interface irb.100;
forwarding-options {
    flood {
        input vlan_100_BUM_flood;
    }
}
```

While not shown, you can also apply a family bridge filter to a bridge domain using the filter keyword. This is where you would apply a bridge family filter to block HTTP within the VLAN, for example. Note that currently only input filters and flood filters are supported for bridge domains.

Verify Proper Operation

Verification begins with confirmation that the filter and policers have been created.

```
{master}[edit]
jnpr@R1-RE0# run show firewall

Filter: vlan_100_BUM_flood
Counters:
Name                                                Bytes              Packets
vlan100_bcast_flood_allowed                             0                    0
vlan100_mcast_flood_allowed                             0                    0
vlan100_unicast_flood_allowed                           0                    0
Policers:
Name                                                Bytes              Packets
vlan100_broadcast-broadcast_flood                       0                    0
vlan100_multicast-mcast_flood                           0                    0
vlan100_unknown_unicast-police_unicast_flood            0                    0

Filter: __default_bpdu_filter__

Filter: discard_vlan100_http_initial
Counters:
Name                                                Bytes              Packets
discard_vlan100_http_initial                            0                    0
```

The output confirms the presence of both the Layer 2 and Layer 3 filters. In the case of the `vlan_100_BUM_flood` filter, all three policers are also shown, and all with the expected 0 counts, given there is full control over traffic sources in the test lab, and at present no user traffic is being generated.

It's time to fix that, so the Router Tester (RT) port attached to S1 is configured to generate 10,000 unicast IP packets to (unassigned) destination IP address 192.0.2.62, using an unassigned destination MAC address to ensure unicast flooding, as this MAC address cannot be learned until it appears as a source MAC address. The traffic generator is set to send 128 byte frames, in a single burst of 10,000 frames, with an average load of 10% and a burst load of 40%.

After sending the traffic the vlan_100_BUM_flood filter is again displayed:

```
{master}[edit firewall]
jnpr@R1-RE0# run show firewall filter vlan_100_BUM_flood

Filter: vlan_100_BUM_flood
Counters:
Name                                                Bytes              Packets
vlan100_bcast_flood_allowed                            64                    1
vlan100_mcast_flood_allowed                           180                    2
vlan100_unicast_flood_allowed                       16000                  125
Policers:
Name                                                Bytes              Packets
vlan100_broadcast-broadcast_flood                       0                    0
vlan100_multicast-mcast_flood                           0                    0
vlan100_unknown_unicast-police_unicast_flood      1303500                 9875
```

The output confirms that policing of unknown unicast has occurred, with a total of 9,875 test frames subjected to the cruel and swift strike of the policer's mighty sword. On a more positive note, the balance of traffic, handed back to the calling term, happens to tally 125 in this run, a value that correlates nicely with the discarded traffic as it exactly matches the 10,000 test frames sent. The two remaining policers are confirmed in the same manner by altering the test traffic to multicast and broadcast, but results are omitted for brevity.

The operation of the HTTP filter is confirmed next. After clearing all filter counters, a Telnet session to port 80 is initiated from S1 to the VIP for VLAN 100, which is assigned IP address 192.0.2.1 in this example. As this traffic is not leaving the VLAN, instead targeting an IP address within the related LIS, you expect no filtering action. Given HTTP services are not enabled at R1, the current master of the VIP for VLAN 100, you do not expect the connection to succeed, either.

First, the expected connectivity to the VIP is confirmed:

```
{master:0}[edit]
jnpr@S1-RE0# run traceroute 192.0.2.1 no-resolve
traceroute to 192.0.2.1 (192.0.2.1), 30 hops max, 40 byte packets
 1  * 192.0.2.1  1.109 ms  4.133 ms

{master:0}[edit]
jnpr@S1-RE0#
```

Then the Telnet session is initiated:

```
{master:0}[edit]
jnpr@S1-RE0# run telnet 192.0.2.1 port 80
Trying 192.0.2.1...
telnet: connect to address 192.0.2.1: Connection refused
telnet: Unable to connect to remote host
```

The connection fails, as expected. The counter confirms there were no filter hits for intra-VLAN traffic:

```
{master}[edit]
jnpr@R1-RE0# run show firewall filter discard_vlan100_tcp_initial

Filter: discard_vlan100_http_initial
Counters:
Name                                                Bytes              Packets
discard_vlan100_http_initial                            0                    0

{master}[edit]
jnpr@R1-RE0#
```

To test inter-VLAN communications, the configuration of S1 is altered to remove its VLAN 200 definition and to add a static default IPv4 route that points to the VLAN 100 VIP address. With these changes, S1 acts like an IP host assigned to VLAN 100 would, using the VLAN's VIP as its default gateway:

```
{master:0}[edit]
jnpr@S1-RE0# run show route 192.0.2.65

inet.0: 9 destinations, 9 routes (9 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

0.0.0.0/0          *[Static/5] 00:00:09
                    > to 192.0.2.1 via vlan.100
```

The target of the Telnet request is now changed to the VIP for the LIS in VLAN 200, the first available IP in the 192.0.2.64 subnet, which is 192.0.2.65. As before, a traceroute is first performed to confirm expected inter-VLAN routing through R1's IRB:

```
{master:0}[edit]
jnpr@S1-RE0# run traceroute 192.0.2.65
traceroute to 192.0.2.65 (192.0.2.65), 30 hops max, 40 byte packets
 1  * 192.0.2.2 (192.0.2.2)  1.089 ms  0.826 ms
 2  192.0.2.65 (192.0.2.65)  4.004 ms  3.137 ms  1.081 ms
```

And now the simulated HTTP connection request, which should be blocked at R1's IRB interface via its input filter:

```
{master:0}[edit]
jnpr@S1-RE0# run telnet 192.0.2.65 port 80
Trying 192.0.2.65...
telnet: connect to address 192.0.2.65: Connection refused
telnet: Unable to connect to remote host

{master:0}[edi
```

Drumroll please . . .

```
{master}[edit interfaces irb]
jnpr@R1-RE0# run show firewall filter discard_vlan100_tcp_initial

Filter: discard_vlan100_http_initial
Counters:
Name                                                Bytes              Packets
discard_vlan100_http_initial                           64                    1
```

The single packet count for the discard_vlan100_http_initial filter confirms the initial TCP SYN segment has met its demise, at least when targeted to the destination port associated with the HTTP service. As a final verification, you telnet over VLAN 100 to the lo0 address of R2, again using inter-VLAN routing; the goal is to confirm that only inter-VLAN HTTP connection requests are caught by the IRB filter:

```
{master:0}[edit]
jnpr@S1-RE0# run telnet 10.3.255.2
Trying 10.3.255.2...
Connected to 10.3.255.2.
Escape character is '^]'.
=============================================================================
Hostname:       r2-MX240
Routing Engine: RE0
=============================================================================
ANNOUNCEMENT
=============================================================================
. . . .
```

As expected, outgoing Telnet sessions are permitted by the filter, which completes verification of the case study.

Summary

The Junos OS combined with Trio-based PFEs offers a rich set of stateless firewall filtering and policing options, and some really cool built-in DDoS capabilities. All are performed in hardware so you can enable them in a scaled production environment without appreciable impact to forwarding rates.

Filters, along with counters and policers, can be used to track customer usage or to enforce SLA contracts that in turn support CoS, all common needs at the edges of a network. Even if you deploy your MX in the core, where these functions are less likely, you still need stateless filters, policers, and/or DDoS protection to protect your router's control plane from unsupported services and to guard against excessive traffic, whether good or bad, to ensure the router remains secure and continues to operate as intended.

This chapter provided current best practice templates from strong RE protection filters for both IPv4 and IPv6 control plane. All readers should compare their current RE protection filters to the examples provided to decide if any modifications are needed to maintain current best practice in this complex, but all too important subject.

Chapter Review Questions

1. Which is true when you omit the `interface-specific` keyword and apply a filter with a counter to multiple interfaces on a single Trio PFE?
    a. Each filter independently counts the traffic on their respective interfaces
    b. This is not supported; `interface-specific` is required to apply the same filter more than once
    c. A single counter will sum the traffic from all interfaces
    d. Both individual filter counters and a single aggregate counter is provided
2. How do you apply a physical interface policer?
    a. With a policer that is marked as a `physical-interface-policer`
    b. Directly to the IFD using the `physcial-interface-policer` statement
    c. By evoking the policer through one or more filters that use the `physical-interface-filter` statement, applied to their respective IFFs on each IFL that is configured on the IFD
    d. Both A and B
    e. Both A and C
3. Which is true regarding a Layer 2 policer?
    a. You cannot evoke via a firewall filter
    b. The policer counts frame overhead, to include the FCS
    c. You can apply at IFL level only, not under a protocol family, using a `layer2-policer` statement, and the policer affects all traffic for all families on that IFL
    d. The policer can be color aware only when applied at egress
    e. All of the above
4. A filter is applied to the main instance lo0.0 and a VRF is defined without its own lo0.n IFL. Which is true?
    a. Traffic from the instance to the local control plane is filtered by the lo0.0 filter
    b. Traffic from the instance to remote VRF destinations is filtered by the lo0.0 filter
    c. Traffic from the instance to the local control plane is not filtered
    d. None of the above. VRFs require an lo0.n for their routing protocols to operate
5. What Junos feature facilitates simplified filter management when using address-based match criteria to permit only explicitly defined BGP peers?
    a. Dynamic filter lists
    b. Prefix-lists and the `apply-path` statement
    c. The ability to specify a 0/0 as a match-all in an address based match condition
    d. All of the above

6. What filter construct permits cascaded policing using single-rate two-color (classical) policers?
    a. Then next-filter flow control action
    b. Then next-term flow control action
    c. By using separate filters as part of a filter list
    d. This is not possible; you must use a hierarchical policer
7. What is needed to control BUM traffic?
    a. An ingress physical interface filter
    b. An egress physical interface filter
    c. A conventional filter or filter/policer combination applied to a Layer 2 instance's Forwarding Table
    d. A sr-TCM policer applied at the unit level for all Layer 2 families using the `layer2-policer` statement
8. The goal is to filter Layer 3 traffic from being sent *into* a bridge-domain on an MX router. Which is the best option?
    a. A Forwarding Table filter in the bridge domain
    b. An input filter on the bridge domain's IRB interface
    c. An output filter on the bridge domain's IRB interface
    d. Output filters on all trunk interfaces that support the bridge domain
9. What is the effect of inducing the `logical-interface-policer` statement in a policer definition?
    a. Allows the policer to be used as a Layer 2 policer
    b. Creates what is called an aggregate policer that acts on all families that share a logical unit
    c. Allows the policer to be used on virtual interfaces like the bridge domain's IRB
    d. Both A and B
10. Which is true regarding three-color policers?
    a. The trTCM uses a single token bucket and best suited to sustained bursts
    b. The srTCM uses a single token bucket and is best suited to sustained bursts
    c. The trTCM uses two token buckets with overflow and supports sustained bursts
    d. The trTCM uses two independent token buckets and supports sustained bursts
    e. The srTCM uses two token buckets with overflow and supports sustained bursts

Chapter Review Answers

1. **Answer: C**. By default, filter and policer stats are aggregated on a per PFE basis, unless you use the interface-specific statement. When the interface-specific keyword is used a separate filter (and if used, policer) instance is created for each application.
2. **Answer: E, both A and C**. A physical interface policer needs the `physical-interface-policer` statement, and you apply via one or more filters that include the `physical-interface-filter statement`, under each family on all the IFLs that share the interface device. The result is a single policer that polices all families on all IFLs.
3. **Answer: E**. All are true regarding a Layer 2 policer, which is also a form of logical interface policer in that it acts on all families that share an IFL.
4. **Answer: A**. When a routing instance has filter applied to an lo0 unit in that instance, that filter is used; otherwise, control plane traffic from the instance to the RE is filtered by the main instance lo0.0 filter.
5. **Answer: B**. You use prefix lists and the apply-path feature to build a dynamic list of prefixes that are defined somewhere else on the router; for example, those assigned to interfaces or used in BGP peer definitions, and then use the dynamic list as a match condition in a filter to simplify filter management in the face of new interface or peer definitions.
6. **Answer: B**. Only with `next-term` can you can have traffic that has been accepted by one policer, which is then returned to the calling term where it's normally implicitly accepted, be forced to fall through to the next term, or filter where it can be subjected to another level of policing. There is no next-filter statement, and simply adding another filter behind the current one does not override the implicit accept that occurs when you use an action modifier like count or police.
7. **Answer: C**. BUM traffic filtering and policing is only possible with a Forwarding Table filter or filter/policer combination. Unknown traffic types cannot be known at ingress, and broadcast cannot be used in an egress interface filter or filter/policer combination.
8. **Answer: C**. Refer to Figure 3-12. Despite the goal of filtering traffic into the bridge domain, the filter goes in the output direction of the IRB interface that serves the VLAN. Traffic from other VLANs or the main instances comes from the Forwarding Table and leaves the IRB to flow into the VLAN. A Forwarding Table filter would impact intra-VLAN communications which is not the requirement.
9. **Answer: D, both A and B**. You create an aggregate policer that operates on all families of a given IFL by using the `logical-interface-policer` statement. A Layer 2 policer is a form of logical interface policer and so also requires this statement.
10. **Answer: D**. Both the single- and two-rate TCM style policers use two buckets. But only in the two-rate version are the buckets independent with no overflow, and the

trTCM is best suited to sustained bursting due to its separate control over committed and peak information rates.

CHAPTER 4

Routing Engine Protection and DDoS Prevention

This chapter builds upon the last by providing a concrete example of stateless firewall filter and policer usage in the context of a routing engine protection filter, and also demonstrates the new Trio-specific DDoS prevention feature that hardens the already robust Junos control plane with no explicit configuration required.

The RE protection topics discussed include:

- IPv4 and IPv6 control plane protection filter case study
- DDoS feature overview
- DDoS protection case study
- Mitigating DDoS with BGP flow-specification
- BGP flow-specification case study

RE Protection Case Study

This case study presents a current best practice example of a stateless filter to protect an MX router's IPv4 and IPv6 control plane. In addition, the recent DDoS detection feature, available on Trio-based MX routers starting with release v11.2, are examined and then combined with RE filtering to harden the router against unauthorized access and resource depletion.

As networks become more critical, security and high availability become ever more crucial. The need for secure access to network infrastructure, both in terms of user-level authentication and authorization and all the way to the configuration and use of secure access protocols like SSH, is a given. So much so, that These topics have been covered in many recent books. So as to not rehash the same information, readers interested in these topics are directed to *Junos Enterprise Routing*, Second Edition, by O'Reilly Media.

The goal of this section is to provide an up-to-date example of a strong RE protection filter for both IPv4 and IPv6, and to address the topic of why basic filters may not guard against resource depletion, which, if allowed to go unchecked, can halt a router's operation just as effectively as any "hacker" who gains unauthorized access to the system with nefarious intent.

The topic of router security is complex and widespread. So much so that informational RFC 6192 was produced to outline IPv4 and IPv6 filtering best practices, along with example filters for both IOS- and Junos OS-based products. There is much overlap between the examples in this section and the RFC's suggestions, which is a good sign, as you can never have too many smart people thinking about security. and It's good to see different approaches and techniques as well as a confirmation that many complex problems have common solutions that have been well tested.

IPv4 RE Protection Filter

This section provides the reader with a current best practice example of an RE protection filter for IPv4 traffic. Protection filters are applied in the input direction to filter traffic arriving on PFE or management ports before it's processed by the RE. Output filters are generally used for CoS marking of locally generated control plane traffic, as opposed to security-related reason as you generally trust your own routers and the traffic they originate. Figure 4-1 provides the topology details that surround this case study.

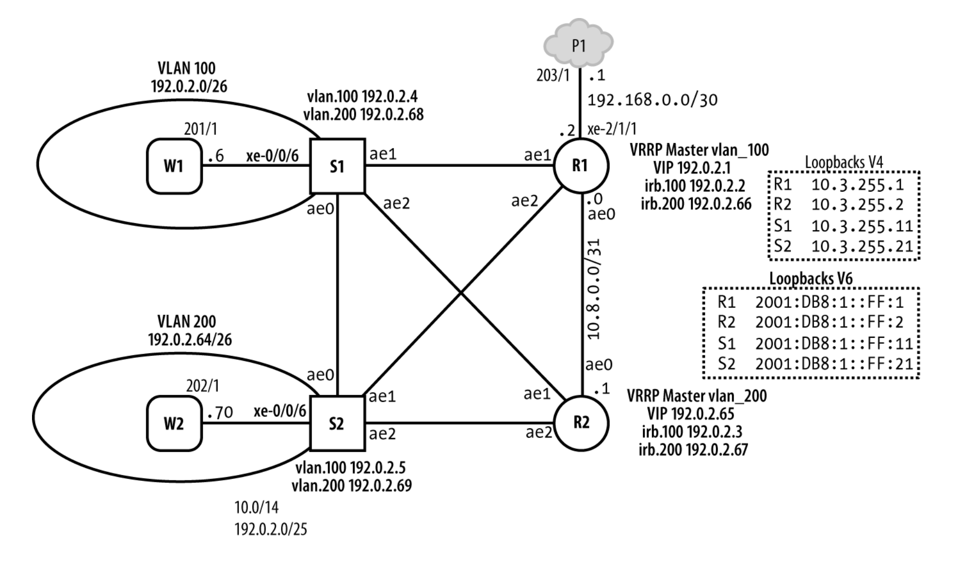


Figure 4-1. DDoS Protection Lab Topology.

The example, used with permission from Juniper Networks Books, is taken from *Day One: Securing the Routing Engine* by Douglas Hanks, also the coauthor of this book.

> Note: Router security is no small matter. The reader is encouraged to examine the filter carefully before adapting it for use in his or her own network.

The principles behind the filter's operation and the specific rationale behind its design framework are explained in the *Day One* book, and so are not repeated here in the interest of brevity. The filter is included here as a case study example for several reasons:

- RE protection is important and needed, and this is a really good filter. There's no point in recreating an already perfectly round wheel, and the *Day One book* is freely available as a PDF.
- The example makes great use of some important Junos features that are not necessarily MX-specific, and so have not been covered in this chapter, including filter nesting (a filter calling another filter), `apply-path`, and `prefix-list`. All are powerful tools that can make managing and understanding a complex filter much simpler. The examples also make use of the `apply-flags omit` statement. This flag results in the related configuration block not being displayed in a `show configuration command`, unless you pipe the results to `display omit`. Again, while not a filter-specific feature, this is another cool Junos capability that can be utilized to make living with long filters that much easier.
- It's a good test of this chapter and the reader's comprehension of the same. This is a real-world example of a complex filter that solves a real issue. While specific protocol nuances, such as the specific multicast addresses used by OSPF, may not be known, having arrived here, the reader should be able to follow the filter's operation and use of policing with little guidance.
- The example is comprehensive, providing support for virtually all known legitimate routing protocols and services; be sure to remove support for any protocols or services that are not currently used, either by deleting the filter in question or by simply not including that filter in the list of filters that you ultimately apply to the lo0 interface. For example, as IS-IS is used in the current lab, there is currently no need for any OSPF-specific filter. Also, be sure to confirm that the prefix lists contain all addresses that should be able to reach the related service or protocol.

When first applying the filter list, you should replace the final `discard-all` term with one that matches all with an accept and log action. This is done as a safeguard to prevent service disruption in the event that a valid service or protocol has not been accommodated by a previous term. After applying the filter, pay special attention to any log hits indicating traffic has made it to the final catch-all term, as this may indicate you have more filter work to do.

> Before applying any RE filter, you should carefully evaluate both the filters/terms and their application order to confirm that all valid services and remote access methods are allowed. In addition, you must also edit the sample prefix list to ensure they accurately reflect *all internal and external addresses* from where the related services should be reachable. Whenever making this type of change, console access should be available in the event that recovery is needed, and you should strongly consider the use of `commit confirmed` command.

When your filter is correctly matched to the particulars of your network, the only traffic that should fall through to the final term should be that which is unsupported and therefore unneeded, and safe to drop. Once it is so confirmed, you should make the `discard-all` filter the last in the chain—its ongoing count and logging actions simplify future troubleshooting when a new service is added and no one can figure out why it's not working. Yes, true security is a pain, but far less so in the long run then the lack of, or worse yet, a false sense of security!

Let's begin with the policy-related configuration where prefix lists are defined in such a way that they automatically populate with addresses assigned to the system itself, as well as well-known addresses associated with common protocols. This small bit of upfront work makes later address-based matches a snap and makes ongoing address and peer definition changes painless, as the filter automatically keeps up. Note that the sample expressions catch all addresses assigned, including those on the management network and GRE tunnels, etc. The sample presumes some use of logical systems (a feature previously known as *logical routers*). Where not applicable you can safely omit the related prefix list.

```
{master}[edit]
regress@R1-RE0# show policy-options | no-more
prefix-list router-ipv4 {
    apply-path "interfaces <*> unit <*> family inet address <*>";
}
prefix-list bgp-neighbors {
    apply-path "protocols bgp group <*> neighbor <*>";
}
prefix-list ospf {
    224.0.0.5/32;
    224.0.0.6/32;
}
prefix-list rfc1918 {
    10.0.0.0/8;
    172.16.0.0/12;
    192.168.0.0/16;
}
prefix-list rip {
    224.0.0.9/32;
}
prefix-list vrrp {
    224.0.0.18/32;
}
```

```
prefix-list multicast-all-routers {
    224.0.0.2/32;
}
prefix-list router-ipv4-logical-systms {
    apply-path "logical-systems <*> interfaces <*> unit <*> family inet address <*>";
}
prefix-list bgp-neighbors-logical-systems {
    apply-path "logical-systems <*> protocols bgp group <*> neighbor <*>";
}
prefix-list radius-servers {
    apply-path "system radius-server <*>";
}
prefix-list tacas-servers {
    apply-path "system tacplus-server <*>";
}
prefix-list ntp-server {
    apply-path "system ntp server <*>";
}
prefix-list snmp-client-lists {
    apply-path "snmp client-list <*> <*>";
}
prefix-list snmp-community-clients {
    apply-path "snmp community <*> clients <*>";
}
prefix-list localhost {
    127.0.0.1/32;
}
prefix-list ntp-server-peers {
    apply-path "system ntp peer <*>";
}
prefix-list dns-servers {
    apply-path "system name-server <*>";
}
```

You can confirm your apply-path and prefix lists are doing what you expect by showing the list and piping the output to `display inheritance`. Again, it's critical that your prefix lists contain all expected addresses from where a service should be reachable, so spending some time here to confirm the regular expressions work as expected is time well spent. Here, the results of the `router-ipv4 apply-path` regular expression are examined.

```
{master}[edit]
jnpr@R1-RE0# show policy-options prefix-list router-ipv4
apply-path "interfaces <*> unit <*> family inet address <*>";

{master}[edit]
jnpr@R1-RE0# show policy-options prefix-list router-ipv4 | display inheritance
##
## apply-path was expanded to:
##     192.168.0.0/30;
##     10.8.0.0/31;
##     192.0.2.0/26;
##     192.0.2.64/26;
##     10.3.255.1/32;
##     172.19.90.0/23;
```

```
##
apply-path "interfaces <*> unit <*> family inet address <*>";
```

If you do not see one or more commented prefixes, as in this example, then either the related configuration does not exist or there is a problem in your path statement. As additional confirmation, consider the sample BGP stanza added to R1, consisting of three BGP peer groups: two IPv6 and one IPv4:

```
{master}[edit]
jnpr@R1-RE0# show protocols bgp
group int_v4 {
    type internal;
    local-address 10.3.255.1;
    neighbor 10.3.255.2;
}
group ebgp_v6 {
    type external;
    peer-as 65010;
    neighbor fd1e:63ba:e9dc:1::1;
}
group int_v6 {
    type internal;
    local-address 2001:db8:1::ff:1;
    neighbor 2001:db8:1::ff:2;
}
```

Once again, the related prefix lists are confirmed to contain all expected entries:

```
{master}[edit]
jnpr@R1-RE0# show policy-options prefix-list bgp-neighbors_v4 | display inheritance
##
## apply-path was expanded to:
##     10.3.255.2/32;
##
apply-path "protocols bgp group <*_v4> neighbor <*>";

{master}[edit]
jnpr@R1-RE0# show policy-options prefix-list bgp-neighbors_v6 | display inheritance
##
## apply-path was expanded to:
##     fd1e:63ba:e9dc:1::1/128;
##     2001:db8:1::ff:2/128;
##
apply-path "protocols bgp group <*_v6> neighbor <*>";
```

And now, the actual filter. It's a long one, but security is never easy and is more an ongoing process than a one-point solution anyway. At least the comprehensive nature of the filter means it's easy to grow into new services or protocols as you simply have to apply the related filters when the new service is turned up:

```
{master}[edit]
jnpr@R1-RE0# show firewall family inet | no-more
prefix-action management-police-set { /* OMITTED */ };
prefix-action management-high-police-set { /* OMITTED */ };
filter accept-bgp { /* OMITTED */ };
filter accept-ospf { /* OMITTED */ };
filter accept-rip { /* OMITTED */ };
filter accept-vrrp { /* OMITTED */ };
filter accept-ssh { /* OMITTED */ };
filter accept-snmp { /* OMITTED */ };
filter accept-ntp { /* OMITTED */ };
filter accept-web { /* OMITTED */ };
filter discard-all { /* OMITTED */ };
filter accept-traceroute { /* OMITTED */ };
filter accept-igp { /* OMITTED */ };
filter accept-common-services { /* OMITTED */ };
filter accept-sh-bfd { /* OMITTED */ };
filter accept-ldp { /* OMITTED */ };
filter accept-ftp { /* OMITTED */ };
filter accept-rsvp { /* OMITTED */ };
filter accept-radius { /* OMITTED */ };
filter accept-tacas { /* OMITTED */ };
filter accept-remote-auth { /* OMITTED */ };
filter accept-telnet { /* OMITTED */ };
filter accept-dns { /* OMITTED */ };
filter accept-ldp-rsvp { /* OMITTED */ };
filter accept-established { /* OMITTED */ };
filter accept-all { /* OMITTED */ };
filter accept-icmp { /* OMITTED */ };
filter discard-frags { /* OMITTED */ };
```

Not much to see, given the omit flag is in play. Easy enough to fix:

```
{master}[edit]
jnpr@R1-RE0# show firewall family inet | no-more | display omit
prefix-action management-police-set {
    apply-flags omit;
    policer management-1m;
    count;
    filter-specific;
    subnet-prefix-length 24;
    destination-prefix-length 32;
}
prefix-action management-high-police-set {
    apply-flags omit;
    policer management-5m;
    count;
    filter-specific;
    subnet-prefix-length 24;
    destination-prefix-length 32;
}
filter accept-bgp {
    apply-flags omit;
    term accept-bgp {
        from {
            source-prefix-list {
                bgp-neighbors_v4;
                bgp-neighbors-logical-systems_v4;
            }
            protocol tcp;
            port bgp;
        }
        then {
            count accept-bgp;
            accept;
        }
    }
}
filter accept-ospf {
    apply-flags omit;
    term accept-ospf {
        from {
            source-prefix-list {
                router-ipv4;
                router-ipv4-logical-systms;
            }
            destination-prefix-list {
                router-ipv4;
                ospf;
                router-ipv4-logical-systms;
            }
            protocol ospf;
        }
        then {
            count accept-ospf;
            accept;
        }
    }
}
filter accept-rip {
    apply-flags omit;
    term accept-rip {
        from {
            source-prefix-list {
                router-ipv4;
                router-ipv4-logical-systms;
            }
            destination-prefix-list {
                rip;
            }
            protocol udp;
            destination-port rip;
        }
        then {
            count accept-rip;
            accept;
        }
    }
    term accept-rip-igmp {
        from {
            source-prefix-list {
                router-ipv4;
                router-ipv4-logical-systms;
            }
            destination-prefix-list {
                rip;
            }
            protocol igmp;
        }
        then {
            count accept-rip-igmp;
            accept;
        }
    }
}
filter accept-vrrp {
    apply-flags omit;
    term accept-vrrp {
        from {
            source-prefix-list {
                router-ipv4;
                router-ipv4-logical-systms;
            }
            destination-prefix-list {
                vrrp;
            }
            protocol [ vrrp ah ];
        }
        then {
            count accept-vrrp;
            accept;
        }
    }
}
filter accept-ssh {
    apply-flags omit;
    term accept-ssh {
        from {
            source-prefix-list {
                rfc1918;
            }
            destination-prefix-list {
                router-ipv4;
                router-ipv4-logical-systms;
            }
            protocol tcp;
            destination-port ssh;
        }
        then {
            policer management-5m;
            count accept-ssh;
            accept;
        }
    }
}
filter accept-snmp {
    apply-flags omit;
    term accept-snmp {
        from {
            source-prefix-list {
                snmp-client-lists;
                snmp-community-clients;
            }
            destination-prefix-list {
                router-ipv4;
                router-ipv4-logical-systms;
            }
            protocol udp;
            destination-port snmp;
        }
        then {
            policer management-5m;
            count accept-snmp;
            accept;
        }
    }
}
filter accept-ntp {
    apply-flags omit;
    term accept-ntp {
        from {
            source-prefix-list {
                ntp-server;
            }
            destination-prefix-list {
                router-ipv4;
                router-ipv4-logical-systms;
            }
            protocol udp;
            port ntp;
        }
        then {
            policer management-1m;
            count accept-ntp;
            accept;
        }
    }
    term accept-ntp-peer {
        from {
            source-prefix-list {
                ntp-server-peers;
            }
            destination-prefix-list {
                router-ipv4;
                router-ipv4-logical-systms;
            }
            protocol udp;
            destination-port ntp;
        }
        then {
            policer management-1m;
            count accept-ntp-peer;
            accept;
        }
    }
    term accept-ntp-server {
        from {
            source-prefix-list {
                rfc1918;
            }
            destination-prefix-list {
                router-ipv4;
                router-ipv4-logical-systms;
            }
            protocol udp;
            destination-port ntp;
        }
        then {
            policer management-1m;
            count accept-ntp-server;
            accept;
        }
    }
}
filter accept-web {
    apply-flags omit;
    term accept-web {
        from {
            source-prefix-list {
                rfc1918;
            }
            destination-prefix-list {
                router-ipv4;
                router-ipv4-logical-systms;
            }
            protocol tcp;
            destination-port [ http https ];
        }
        then {
            policer management-5m;
            count accept-web;
            accept;
        }
    }
}
filter discard-all {
    apply-flags omit;
    term discard-ip-options {
        from {
            ip-options any;
        }
        then {
            count discard-ip-options;
            log;
            syslog;
            discard;
        }
    }
    term discard-TTL_1-unknown {
        from {
            ttl 1;
        }
        then {
            count discard-all-TTL_1-unknown;
            log;
            syslog;
            discard;
        }
    }
    term discard-tcp {
        from {
            protocol tcp;
        }
        then {
            count discard-tcp;
            log;
            syslog;
            discard;
        }
    }
    term discard-netbios {
        from {
            protocol udp;
            destination-port 137;
        }
        then {
            count discard-netbios;
            log;
            syslog;
            discard;
        }
    }
    term discard-udp {
        from {
            protocol udp;
        }
        then {
            count discard-udp;
            log;
            syslog;
            discard;
        }
    }
    term discard-icmp {
        from {
            protocol icmp;
        }
        then {
            count discard-icmp;
            log;
            syslog;
            discard;
        }
    }
    term discard-unknown {
        then {
```

```
            count discard-unknown;
            log;
            syslog;
            discard;
        }
    }
}
filter accept-traceroute {
    apply-flags omit;
    term accept-traceroute-udp {
        from {
            destination-prefix-list {
                router-ipv4;
                router-ipv4-logical-systms;
            }
            protocol udp;
            ttl 1;
            destination-port 33435-33450;
        }
        then {
            policer management-1m;
            count accept-traceroute-udp;
            accept;
        }
    }
    term accept-traceroute-icmp {
        from {
            destination-prefix-list {
                router-ipv4;
                router-ipv4-logical-systms;
            }
            protocol icmp;
            ttl 1;
            icmp-type [ echo-request timestamp time-exceeded ];
        }
        then {
            policer management-1m;
            count accept-traceroute-icmp;
            accept;
        }
    }
    term accept-traceroute-tcp {
        from {
            destination-prefix-list {
                router-ipv4;
                router-ipv4-logical-systms;
            }
            protocol tcp;
            ttl 1;
        }
        then {
            policer management-1m;
            count accept-traceroute-tcp;
            accept;
        }
    }
}
filter accept-igp {
    apply-flags omit;
    term accept-ospf {
        filter accept-ospf;
    }
    term accept-rip {
        filter accept-rip;
    }
}
filter accept-common-services {
    apply-flags omit;
    term accept-icmp {
        filter accept-icmp;
    }
    term accept-traceroute {
        filter accept-traceroute;
    }
    term accept-ssh {
        filter accept-ssh;
    }
    term accept-snmp {
        filter accept-snmp;
    }
    term accept-ntp {
        filter accept-ntp;
    }
    term accept-web {
        filter accept-web;
    }
    term accept-dns {
        filter accept-dns;
    }
}
filter accept-sh-bfd {
    apply-flags omit;
    term accept-sh-bfd {
        from {
            source-prefix-list {
                router-ipv4;
                router-ipv4-logical-systms;
            }
            destination-prefix-list {
                router-ipv4;
                router-ipv4-logical-systms;
            }
            protocol udp;
            source-port 49152-65535;
            destination-port 3784-3785;
        }
        then {
            count accept-sh-bfd;
            accept;
        }
    }
}
filter accept-ldp {
    apply-flags omit;
    term accept-ldp-discover {
        from {
            source-prefix-list {
                router-ipv4;
                router-ipv4-logical-systms;
            }
            destination-prefix-list {
                multicast-all-routers;
            }
            protocol udp;
            destination-port ldp;
        }
        then {
            count accept-ldp-discover;
            accept;
        }
    }
    term accept-ldp-unicast {
        from {
            source-prefix-list {
                router-ipv4;
                router-ipv4-logical-systms;
            }
            destination-prefix-list {
                router-ipv4;
                router-ipv4-logical-systms;
            }
            protocol tcp;
            port ldp;
        }
        then {
            count accept-ldp-unicast;
            accept;
        }
    }
    term accept-tldp-discover {
        from {
            destination-prefix-list {
                router-ipv4;
                router-ipv4-logical-systms;
            }
            protocol udp;
            destination-port ldp;
        }
        then {
            count accept-tldp-discover;
            accept;
        }
    }
    term accept-ldp-igmp {
        from {
            source-prefix-list {
                router-ipv4;
                router-ipv4-logical-systms;
            }
            destination-prefix-list {
                multicast-all-routers;
            }
            protocol igmp;
        }
        then {
            count accept-ldp-igmp;
            accept;
        }
    }
}
filter accept-ftp {
    apply-flags omit;
    term accept-ftp {
        from {
            source-prefix-list {
                rfc1918;
            }
            destination-prefix-list {
                router-ipv4;
                router-ipv4-logical-systms;
            }
            protocol tcp;
            port [ ftp ftp-data ];
        }
        then {
            policer management-5m;
            count accept-ftp;
            accept;
        }
    }
}
filter accept-rsvp {
    apply-flags omit;
    term accept-rsvp {
        from {
            destination-prefix-list {
                router-ipv4;
                router-ipv4-logical-systms;
            }
            protocol rsvp;
        }
        then {
            count accept-rsvp;
            accept;
        }
    }
}
filter accept-radius {
    apply-flags omit;
    term accept-radius {
        from {
            source-prefix-list {
                radius-servers;
            }
            destination-prefix-list {
                router-ipv4;
                router-ipv4-logical-systms;
            }
            protocol udp;
            source-port [ radacct radius ];
            tcp-established;
        }
        then {
            policer management-1m;
            count accept-radius;
            accept;
        }
    }
}
filter accept-tacas {
    apply-flags omit;
    term accept-tacas {
        from {
            source-prefix-list {
                tacas-servers;
            }
            destination-prefix-list {
                router-ipv4;
                router-ipv4-logical-systms;
            }
            protocol [ tcp udp ];
            source-port [ tacacs tacacs-ds ];
            tcp-established;
        }
        then {
            policer management-1m;
            count accept-tacas;
            accept;
        }
    }
}
filter accept-remote-auth {
    apply-flags omit;
    term accept-radius {
        filter accept-radius;
    }
    term accept-tacas {
        filter accept-tacas;
    }
}
filter accept-telnet {
    apply-flags omit;
    term accept-telnet {
        from {
            source-prefix-list {
                rfc1918;
            }
            destination-prefix-list {
                router-ipv4;
                router-ipv4-logical-systms;
            }
            protocol tcp;
            destination-port telnet;
        }
        then {
            policer management-1m;
            count accept-telnet;
            accept;
        }
    }
}
filter accept-dns {
    apply-flags omit;
    term accept-dns {
        from {
            source-prefix-list {
                dns-servers;
            }
            destination-prefix-list {
                router-ipv4;
                router-ipv4-logical-systms;
            }
            protocol [ udp tcp ];
            source-port 53;
        }
        then {
            policer management-1m;
            count accept-dns;
            accept;
        }
    }
}
filter accept-ldp-rsvp {
    apply-flags omit;
    term accept-ldp {
        filter accept-ldp;
    }
    term accept-rsvp {
        filter accept-rsvp;
    }
}
filter accept-established {
    apply-flags omit;
    term accept-established-tcp-ssh {
        from {
            destination-prefix-list {
                router-ipv4;
                router-ipv4-logical-systms;
            }
            source-port ssh;
            tcp-established;
        }
        then {
            policer management-5m;
            count accept-established-tcp-ssh;
            accept;
        }
    }
    term accept-established-tcp-ftp {
        from {
            destination-prefix-list {
                router-ipv4;
                router-ipv4-logical-systms;
            }
            source-port ftp;
            tcp-established;
        }
        then {
            policer management-5m;
            count accept-established-tcp-ftp;
            accept;
        }
    }
    term accept-established-tcp-ftp-data-syn {
        from {
            destination-prefix-list {
                router-ipv4;
                router-ipv4-logical-systms;
            }
            source-port ftp-data;
            tcp-initial;
        }
        then {
            policer management-5m;
            count accept-established-tcp-ftp-data-syn;
            accept;
        }
    }
    term accept-established-tcp-ftp-data {
        from {
            destination-prefix-list {
                router-ipv4;
                router-ipv4-logical-systms;
            }
            source-port ftp-data;
            tcp-established;
        }
        then {
            policer management-5m;
            count accept-established-tcp-ftp-data;
            accept;
        }
    }
    term accept-established-tcp-telnet {
        from {
```

```
                destination-prefix-list {
                    router-ipv4;
                    router-ipv4-logical-systms;
                }
                source-port telnet;
                tcp-established;
            }
            then {
                policer management-5m;
                count accept-established-tcp-telnet;
                accept;
            }
        }
        term accept-established-tcp-fetch {
            from {
                destination-prefix-list {
                    router-ipv4;
                    router-ipv4-logical-systms;
                }
                source-port [ http https ];
                tcp-established;
            }
            then {
                policer management-5m;
                count accept-established-tcp-fetch;
                accept;
            }
        }
        term accept-established-udp-ephemeral {
            from {
                destination-prefix-list {
                    router-ipv4;
                    router-ipv4-logical-systms;
                }
                protocol udp;
                destination-port 49152-65535;
            }
            then {
                policer management-5m;
                count accept-established-udp-ephemeral;
                accept;
            }
        }
    }
    filter accept-all {
        apply-flags omit;
        term accept-all-tcp {
            from {
                protocol tcp;
            }
            then {
                count accept-all-tcp;
                log;
                syslog;
                accept;
            }
        }
        term accept-all-udp {
            from {
                protocol udp;
            }
            then {
                count accept-all-udp;
                log;
                syslog;
                accept;
            }
        }
        term accept-all-igmp {
            from {
                protocol igmp;
            }
            then {
                count accept-all-igmp;
                log;
                syslog;
                accept;
            }
        }
        term accept-icmp {
            from {
                protocol icmp;
            }
            then {
                count accept-all-icmp;
                log;
                syslog;
                accept;
            }
        }
        term accept-all-unknown {
            then {
                count accept-all-unknown;
                log;
                syslog;
                accept;
            }
        }
    }
    filter accept-icmp {
        apply-flags omit;
        term no-icmp-fragments {
            from {
                is-fragment;
                protocol icmp;
            }
            then {
                count no-icmp-fragments;
                log;
                discard;
            }
        }
        term accept-icmp {
            from {
                protocol icmp;
                ttl-except 1;
              icmp-type [ echo-reply echo-request time-exceeded unreachable source-quench
    router-advertisement parameter-problem ];
            }
            then {
                policer management-5m;
                count accept-icmp;
                accept;
            }
        }
    }
    filter discard-frags {
        term 1 {
            from {
                first-fragment;
            }
            then {
                count deny-first-frags;
                discard;
            }
        }
        term 2 {
            from {
                is-fragment;
            }
            then {
                count deny-other-frags;
                discard;
            }
        }
    }
```

After all that work, don't forget to actually apply all applicable filters as an `input-list` under family `inet` on the lo0 interface. Before making any changes, please carefully consider the following suggestions, however:

Before actually activating the lo0 application of the IPv4 protection filter, you should:

a. Confirm that all prefix lists are accurate for your networks and that they encompass the necessary address ranges.

b. Confirm that all valid services and remote access protocols are accepted in a filter, and that the filter is included in the input list; for example, in *Day One: Securing The Routing Engine*, the `accept-telnet` filter is not actually applied because Telnet is a nonsecure protocol, and frankly should never be used in a production network. While Telnet is used to access the testbed needed to develop this material, making the absence of the `accept-telnet` filter pretty obvious at time of commit . . . don't ask me how I know this.

c. Make sure the filter initially ends in a match-all term with accept and log actions to make sure no valid services are denied.

d. Consider using commit confirmed for this type of change. Again, don't ask me how I know, but there is a hint in the preceding paragraphs.

The final RE protection filters used in this case study were modified from the example used in the *Day One* book in the following ways:

- The accept-telnet filter is applied in the list; as a lab, Telnet is deemed acceptable. The OSPF and RIP filters are omitted as not in use or planned in the near future.
- The `accept-icmp` filter is modified to no longer match on fragments; this function is replaced with a global deny fragments filter that's applied at the front of the filter list. See the related sidebar.

The list of filters applied to the lo0 interface of R1 for this example is shown; note that the list now begins with the `discard-frags` filter, the inclusion of the `accept-telnet` filter, and that the final `discard-all` filter is in effect. Again, for initial application in a production network, consider using a final match-all filter with accept and log actions to first confirm that no valid services are falling through to the final term before switching over to a final discard action.

Filters and Fragments

Stateless filters with upper-layer protocol match criteria have problems with fragments. And there is no real solution if you insist on a stateless filter. You must choose the lesser of two evils: either deny all fragments up front or do the opposite and accept them all, again, right up front. The former only works when your network's MTUs are properly architected, such that fragmentation of internal traffic simply does not happen.

To see the issue, consider a filter designed to match on ICMP messages with type echo request along with an accept function. Now, imagine that a user generates large ICMP messages with some form of evil payload, and that each message is fragmented into four smaller packets. The issue is that only the first fragment will contain the ICMP header along with its message type code. The remaining three fragments have a copy of the original IP header, with adjusted fragmentation fields, and a payload that simply picks up where the previous fragment left off. This means that only the first fragment is able to be reliably matched against an ICMP message type. The filter code will attempt to match the presumed ICMP header in the remaining three fragments, but recall there is no header present, and this can lead to unpredictable results. In the common case, the fragment will not match the filter and be discarded, which makes the first fragment useless and a waste of host buffer space, as its reassembly timer runs in earnest for fragments that have long since met their demise. In the less likely case, some fragment's payload may match a valid ICMP message type and be accepted in a case where the first fragment, with a valid ICMP header, was discarded.

The Junos OS supports bit field and text alias matching on fragmentation fields for IPv4 and on the fragmentation extension header for IPv6. For example, the `is-fragment` keyword specifies all but the first fragment of an IP packet and is equal to a bit-field match against IPv4 with the condition being `fragment-offset 0 except`, which indicates a trailing fragment of a fragmented packet. You use the `first-fragment` keyword to nix the beginning of the chain, and both are typically used together to either deny or accept all fragments in the first filter term.

The filter does not include the `allow-ospf` or allow-rip filters as the current test bed is using IS-IS, which cannot be affected by an `inet` family filter anyway. It's worth noting that the `accept-sh-bfd` filter is so named as the port range specified allows single-hop BFD sessions only. According to draft-ietf-bfd-multihop-09.txt (now RFC 5883), multihop BFD sessions must use UDP destination port 4784.

```
{master}[edit]
regress@R1-RE0# show interfaces lo0
unit 0 {
    family inet {
        filter {
            input-list [ discard-frags accept-common-services accept-sh-bfd accept-bgp
              accept-ldp accept-rsvp accept-telnet discard-all ];
        }
        address 10.3.255.1/32;
    }
    family iso {
        address 49.0001.0100.0325.5001.00;
    }
    family inet6 {
        address 2001:db8:1::ff:1/128;
    }
}
```

A syslog is added to catch and consolidate any filter-related syslog actions for easy debug later. Remember, the log action writes to kernel cache that is overwritten and lost in a reboot, while syslog can support file archiving and remote logging. Here, the local syslog is configured:

```
jnpr@R1-RE0# show system syslog
file re_filter {
    firewall any;
    archive size 10m;
}
```

After committing the filter, and breathing a sigh of relief as you confirm that remote access is still working (this time), let's quickly look for any issues. To begin with, filter application is confirmed:

```
{master}[edit]
jnpr@R1-RE0# run show interfaces filters lo0
Interface       Admin Link Proto Input Filter        Output Filter
lo0             up    up
lo0.0           up    up   inet  lo0.0-i
                           iso
                           inet6
lo0.16384       up    up   inet
lo0.16385       up    up   inet
```

Next, examine the syslog to see what traffic is falling through unmatched to be discarded:

```
{master}[edit]
jnpr@R1-RE0# run show log re_filter
Dec 12 12:58:09  R1-RE0 fpc2 PFE_FW_SYSLOG_IP: FW: irb.200
                  D vrrp 192.0.2.67 224.0.0.18   0  0  (1 packets)
Dec 12 12:58:15  R1-RE0 last message repeated 7 times
Dec 12 12:58:16  R1-RE0 fpc2 PFE_FW_SYSLOG_IP: FW: irb.200
                  D vrrp 192.0.2.67 224.0.0.18   0  0  (2 packets)
Dec 12 12:58:17  R1-RE0 fpc2 PFE_FW_SYSLOG_IP: FW: irb.200
                  D vrrp 192.0.2.67 224.0.0.18   0  0  (1 packets)
Dec 12 12:58:21  R1-RE0 last message repeated 4 times
Dec 12 12:58:22  R1-RE0 fpc2 PFE_FW_SYSLOG_IP: FW: irb.200
                  D vrrp 192.0.2.67 224.0.0.18   0  0  (2 packets)
Dec 12 12:58:23  R1-RE0 fpc2 PFE_FW_SYSLOG_IP: FW: irb.200
                  D vrrp 192.0.2.67 224.0.0.18   0  0  (1 packets)
Dec 12 12:58:26  R1-RE0 last message repeated 3 times
Dec 12 12:58:27  R1-RE0 fpc2 PFE_FW_SYSLOG_IP: FW: irb.200
                  D vrrp 192.0.2.67 224.0.0.18   0  0  (2 packets)
Dec 12 12:58:28  R1-RE0 fpc2 PFE_FW_SYSLOG_IP: FW: irb.200
                  D vrrp 192.0.2.67 224.0.0.18   0  0  (1 packets)
```

Do'h! What was that warning about confirming the applied filter has support for all supported services, and about using an accept-all in the final term until proper operating is confirmed, again? The syslog action in the final `discard-all` filter has quickly shown that VRRP is being denied by the filter, which readily explains why VRRP is down, and the phones are starting to ring. The applied filter list is modified by adding the `accept-vrrp` filter; note the use of the `insert` function to ensure the correct ordering of filters by making sure that the `discard-all` filter remains at the end of the list:

```
{master}[edit interfaces lo0 unit 0 family inet]
jnpr@R1-RE0# set filter input-list accept-vrrp

{master}[edit interfaces lo0 unit 0 family inet]
jnpr@R1-RE0# show
filter {
    input-list [ discard-frags accept-common-services accept-sh-bfd accept-bgp
      accept-ldp accept-rsvp accept-telnet discard-all accept-vrrp ];
}
address 10.3.255.1/32;

{master}[edit interfaces lo0 unit 0 family inet]
jnpr@R1-RE0# insert filter input-list accept-vrrp before discard-all

{master}[edit interfaces lo0 unit 0 family inet]
jnpr@R1-RE0# show
filter {
    input-list [ discard-frags accept-common-services accept-ospf accept-rip
      accept-sh-bfd accept-bgp accept-ldp accept-rsvp accept-telnet accept-vrrp
        discard-all ];
}
address 10.3.255.1/32;
```

After the change the log file is cleared, and after a few moments redisplayed:

```
{master}[edit interfaces lo0 unit 0 family inet]
jnpr@R1-RE0# run clear log re_filter

. . .
{master}[edit interfaces lo0 unit 0 family inet]
jnpr@R1-RE0# run show log re_filter
Dec 12 13:09:59 R1-RE0 clear-log[21857]: logfile cleared

{master}[edit interfaces lo0 unit 0 family inet]
jnpr@R1-RE0#
```

Perfect—the lack of syslog entry and continued operation of existing services confirms proper operation of the IPv4 RE protection filter.

IPv6 RE Protection Filter

While we have IPv4 running, many networks are only now beginning to deploy IPv6. Given the lack of ubiquity, IPv6 control planes have not been the target of many attacks; many operators have not felt the need to deploy IPv6 RE protection, leading to a general lack of experience in IPv6 filtering best practices.

Next-Header Nesting, the Bane of Stateless Filters

A significant issue with any IPv6 filtering scheme is IPv6's use of next-header nesting, which makes some stateless filtering tasks tricky, if not downright impossible. IPv6, as defined in RFC 2460, states: "In IPv6, optional internet-layer information is encoded in separate headers that may be placed between the IPv6 header and the upper-layer header in a packet. . . . an IPv6 packet may carry zero, one, or more extension headers, each identified by the Next Header field of the preceding header."

The net result is that there can be multiple extension headers placed between the IPv6 header and the upper layer protocol that you might want to match on (TCP, UDP, OSPF3, ICMP6, etc.). Stateless filters are designed to extract keys for matching packet fields using bit positions within a packet that are assumed to be found in the same location. Stateless IPv6 filters on Trio are able to match on the first protocol (next header) that is identified in the IPv6 packet's next-header field, and/or on bits within the actual payload, i.e., the transport protocol (TCP or UDP) ports. There is no flittering capability based on the actual contents of any extension header, and that in the 11.4 release you cannot match on the payload, for example to match a TCP port, when any extension header is present. The ability to match both the first extension header and a payload port is expected in a future release.

> However, regardless of how many extension headers are present, Trio ASICs have the ability to extract the first 32 bits following the last extension header to facilitate Layer 4 (TCP or UDP) port-based matches, even when one or more extension headers are present. On a supported release, the ability to match on a payload protocol when extension headers are present is enabled by specifying the `payload-protocol` keyword in your match criteria.

The presence of extension headers leads to unpredictable filter operation when using a `next-header` match condition. For example, consider a user who wants to filter out Multicast Listener Discovery messages (MLD does for IPv6 what IGMP does for IPv4: it allows multicast hosts to express interest in listening to a multicast group). The user knows that that MLD is an extension of ICMP6, and happily proceeds to create (and commit) the filter shown, only to find MLD messages are not matched, and therefore still allowed to pass through the filter:

```
{master}[edit firewall family inet6]
jnpr@R1-RE0# show
filter count_mld {
    term 1 {
        from {
            next-header icmp;
            icmp-type [ membership-query membership-report membership-termination ];
        }
        then {
            count mld_traffic;
            discard;
        }
    }
    term 2 {
        then accept;
    }
}
```

In this case, a quick look at RFC for MLD (RFC 3810) and the previous restriction on being able to match on a single next-header makes the reason for the filter's failure clear. MLD requires the inclusion of the hop-by-hop extension header, which must precede the ICMP6 header that the filter seeks to match. This means if you want to filter MLD using a stateless filter you must, in fact, set the filter to match on the presence of the hop-by-hop header rather the header you really wanted. The obvious issue here is that other protocols, like RSVP, can also make use of a hop-by-hop header (though Junos does not currently support IPv6-based MPLS signaling), so wholesale filtering based on hop-by-hop (or other extension) headers can lead to unexpected filtering actions.

MLD and the Hop-by-Hop Header

The hop-by-hop header is required for MLD in order to convey the Router Alert (RA) option. The RA function is used to force a router to process the following message even though the packet is not addressed to the router and would otherwise be of no interest. Here, the router may not be an interested listener in the multicast group that is being joined (or left), and therefore might not process the MLD message if not for the RA function.

The Sample IPv6 Filter

As with the IPv4 filter example, it's assumed that the reader is familiar with Junos firewall filter syntax and operation, as well basic IPv6 protocol operation, header fields, and option extension headers. As always, when it comes to filters, no one size fits all, and the reader is encouraged to carefully consider the effects of the sample filter along with careful testing of its operation against the specific IPv6 protocols supported in their networks so that any necessary adjustments can be made before being placed into use on a production network.

Additional details on IPv6 protocol filtering specific to the broad range of possible ICMPv6 message types can be found in RFC 4890, "Recommendations for Filtering ICMPv6 Messages in Firewalls."

To begin, the IPv6 prefix list definitions are displayed; the previous lists used for IPv4 remain in place, with the exception noted in the following:

```
jnpr@R1-RE0# show policy-options
prefix-list router-ipv4 {
. . .
prefix-list bgp-neighbors_v4 {
    apply-path "protocols bgp group <*_v4> neighbor <*>";
}
prefix-list router-ipv6 {
    apply-path "interfaces <*> unit <*> family inet6 address <*>";
}
prefix-list bgp-neighbors_v6 {
    apply-path "protocols bgp group <*_v6> neighbor <*>";
}
prefix-list link_local {
    fe80::/64;
}
prefix-list rfc3849 {
    2001:db8::/32;
}
```

The IPv6-based prefix list performs the same function as their V4 counterparts. IPv6's use of Link Local addressing for many routing protocols means you need to include support for them, as well as your global IPv6 interface routes. Note that the previous `bgp-neighbors` prefix list, as originally used for IPv4, has been renamed and the `apply-path` regular expression modified, so as to not conflict with the same function in IPv6. This approach assumes that you place IPv4 and IPv6 peers in separate groups with a group name that ends in either `_v4` or `_v6`. The IPv6 RE protection filters are displayed:

```
{master}[edit firewall family inet6]
jnpr@R1-RE0#
filter discard-extension-headers {
    apply-flags omit;
    term discard-extension-headers {
        from {
            # Beware - VRRPv3 with authentication or OSPFv3 with Authentication
                enabled may use AH/ESP!
            next-header [ ah dstopts egp esp fragment gre icmp igmp ipip ipv6
                no-next-header routing rsvp sctp ];
        }
        then {
            count discard-ipv6-extension-headers;
            log;
            syslog;
            discard;
        }
    }
}
filter deny-icmp6-undefined {
    apply-flags omit;
    term icmp6-unassigned-discard {
        from {
            next-header icmpv6;
            icmp-type [ 102-106 155-199 202-254 ];
        }
        then discard;
    }
    term rfc4443-discard {
        from {
            next-header icmpv6;
            icmp-type [ 100-101 200-201 ];
        }
        then discard;
    }
}
filter accept-icmp6-misc {
    apply-flags omit;
    term neigbor-discovery-accept {
        from {
            next-header icmpv6;
            icmp-type 133-136;
        }
        then accept;
    }
    term inverse-neigbor-discovery-accept {
        from {
            next-header icmpv6;
            icmp-type 141-142;
        }
        then accept;
    }
    term icmp6-echo-request {
        from {
            next-header icmpv6;
```

```
                icmp-type echo-request;
            }
            then accept;
        }
        term icmp6-echo-reply {
            from {
                next-header icmpv6;
                icmp-type echo-reply;
            }
            then accept;
        }
        term icmp6-dest-unreachable-accept {
            from {
                next-header icmpv6;
                icmp-type destination-unreachable;
            }
            then accept;
        }
        term icmp6-packet-too-big-accept {
            from {
                next-header icmpv6;
                icmp-type packet-too-big;
            }
            then accept;
        }
        term icmp6-time-exceeded-accept {
            from {
                next-header icmpv6;
                icmp-type time-exceeded;
                icmp-code 0;
            }
            then accept;
        }
        term icmp6-parameter-problem-accept {
            from {
                next-header icmpv6;
                icmp-type parameter-problem;
                icmp-code [ 1 2 ];
            }
            then accept;
        }
    }
    filter accept-shsh-bfd-v6 {
        apply-flags omit;
        term accept-sh-bfd-v6 {
            from {
                source-prefix-list {
                    router-ipv6;
                }
                destination-prefix-list {
                    router-ipv6;
                }
                source-port 49152-65535;
                destination-port 3784-3785;
            }
            then accept;
        }
    }
    filter accept-MLD-hop-by-hop_v6 {
        apply-flags omit;
        term bgp_v6 {
            from {
                next-header hop-by-hop;
            }
            then {
                count hop-by-hop-extension-packets;
                accept;
            }
        }
    }
    filter accept-bgp-v6 {
        apply-flags omit;
        term bgp_v6 {
            from {
                prefix-list {
                    rfc3849;
                    bgp-neighbors_v6;
                }
                next-header tcp;
                destination-port bgp;
            }
            then accept;
        }
    }
    filter accept-ospf3 {
        apply-flags omit;
        term ospfv3 {
            from {
                source-prefix-list {
                    link_local;
                }
                next-header ospf;
            }
            then accept;
        }
    }
    filter accept-dns-v6 {
        apply-flags omit;
        term dnsv6 {
            from {
                source-prefix-list {
                    rfc3849;
                }
                next-header [ udp tcp ];
                port domain;
            }
            then accept;
        }
    }
    filter accept-ntp-v6 {
        apply-flags omit;
        term ntpv6 {
            from {
                source-prefix-list {
                    rfc3849;
                }
                next-header udp;
                destination-port ntp;
            }
            then accept;
        }
    }
    filter accept-ssh-v6 {
        apply-flags omit;
        term sshv6 {
            from {
                source-prefix-list {
                    rfc3849;
                }
                next-header tcp;
                destination-port ssh;
            }
            then {
                policer management-5m;
                count accept-ssh;
                accept;
            }
        }
    }
    filter accept-snmp-v6 {
        apply-flags omit;
        term snmpv6 {
            from {
                source-prefix-list {
                    rfc3849;
                }
                next-header udp;
                destination-port snmp;
            }
            then accept;
        }
    }
    filter accept-radius-v6 {
        apply-flags omit;
        term radiusv6 {
            from {
                source-prefix-list {
                    rfc3849;
                }
                next-header udp;
                port [ 1812 1813 ];
            }
            then accept;
        }
    }
```

```
filter accept-telnet-v6 {
    apply-flags omit;
    term telnetv6 {
        from {
            source-prefix-list {
                rfc3849;
            }
            next-header tcp;
            port telnet;
        }
        then {
            policer management-5m;
            count accept-ssh;
            accept;
        }
    }
}
filter accept-common-services-v6 {
    apply-flags omit;
    term accept-icmp6 {
        filter accept-icmp6-misc;
    }
    term accept-traceroute-v6 {
        filter accept-traceroute-v6;
    }
    term accept-ssh-v6 {
        filter accept-ssh-v6;
    }
    term accept-snmp-v6 {
        filter accept-snmp-v6;
    }
    term accept-ntp-v6 {
        filter accept-ntp-v6;
    }
    term accept-dns-v6 {
        filter accept-dns-v6;
    }
}
filter accept-traceroute-v6 {
    apply-flags omit;
    term accept-traceroute-udp {
        from {
            destination-prefix-list {
                router-ipv6;
            }
            next-header udp;
            destination-port 33435-33450;
            hop-limit 1;
        }
        then {
            policer management-1m;
            count accept-traceroute-udp-v6;
            accept;
        }
    }
```

```
        term accept-traceroute-icmp6 {
            from {
                destination-prefix-list {
                    router-ipv6;
                }
                next-header icmp;
                icmp-type [ echo-request time-exceeded ];
                hop-limit 1;
            }
            then {
                policer management-1m;
                count accept-traceroute-icmp6;
                accept;
            }
        }
        term accept-traceroute-tcp-v6 {
            from {
                destination-prefix-list {
                    router-ipv6;
                }
                next-header tcp;
                hop-limit 1;
            }
            then {
                policer management-1m;
                count accept-traceroute-tcp-v6;
                accept;
            }
        }
    }
    filter discard-all-v6 {
        apply-flags omit;
        term discard-HOPLIMIT_1-unknown {
            from {
                hop-limit 1;
            }
            then {
                count discard-all-HOPLIMIT_1-unknown;
                log;
                syslog;
                discard;
            }
        }
        term discard-tcp-v6 {
            from {
                next-header tcp;
            }
            then {
                count discard-tcp-v6;
                log;
                syslog;
                discard;
            }
        }
        term discard-netbios-v6 {
            from {
                next-header udp;
                destination-port 137;
            }
            then {
                count discard-netbios-v6;
                log;
                syslog;
                discard;
            }
        }
        term discard-udp {
            from {
                next-header udp;
            }
            then {
                count discard-udp-v6;
                log;
                syslog;
                discard;
            }
        }
        term discard-icmp6 {
            from {
                next-header icmp;
            }
            then {
                count discard-icmp;
                log;
                syslog;
                discard;
            }
        }
        term discard-unknown {
            then {
                count discard-unknown;
                log;
                syslog;
                discard;
            }
        }
    }
```

The IPv6 filters makes use of the same policers defined previously for IPv4, and follows the same general modular approach, albeit with less counting actions in terms that accept traffic, their use already being demonstrated for IPv4. In this case, the `discard-extension-headers` filter discards all unused extension headers, including the fragmentation header, which ensures fragments are not subjected to any additional term processing where unpredictable results could occur given a fragment's lack of a transport header. As per the filter's comment, the discard action includes traffic with either the AH and/or EH authentications headers, which can be used for legitimate traffic like OSPF3. As always, you need to carefully gauge the needs of each network against any sample filter and make adjustments accordingly.

As before, the relevant list of IPv6 filters are again applied as an input list to the lo0 interface. Now under family inet6:

```
{master}[edit]
jnpr@R1-RE0# show interfaces lo0 unit 0
family inet {
    filter {
        input-list [ discard-frags accept-common-services accept-sh-bfd accept-bgp
          accept-ldp accept-rsvp accept-telnet accept-vrrp discard-all ];
    }
    address 10.3.255.1/32;
}
family iso {
    address 49.0001.0100.0325.5001.00;
}
family inet6 {
    filter {
        input-list [ discard-extension-headers accept-MLD-hop-by-hop_v6
         deny-icmp6-undefined accept-common-services-v6 accept-sh-bfd-v6 accept-bgp-v6
          accept-telnet-v6 accept-ospf3 accept-radius-v6 discard-all-v6 ];
    }
    address 2001:db8:1::ff:1/128;
}
```

After applying the IPv6 filter, the syslog is cleared; after a few moments, it's possible to display any new matches. Recall that at this stage only unauthorized traffic should be reaching the final discard-all action for both the IPv4 and IPv6 filter lists:

```
{master}[edit]
jnpr@R1-RE0# run show log re_filter
Dec 13 10:26:51 R1-RE0 clear-log[27090]: logfile cleared
Dec 13 10:26:52  R1-RE0 /kernel: FW: fxp0.0  D  tcp 172.17.13.146 172.19.90.172
  34788  21
Dec 13 10:26:55  R1-RE0 /kernel: FW: fxp0.0  D  tcp 172.17.13.146 172.19.90.172
  34788  21
Dec 13 10:26:55  R1-RE0 /kernel: FW: fxp0.0  D igmp 172.19.91.95 224.0.0.1  0  0
Dec 13 10:27:01  R1-RE0 /kernel: FW: fxp0.0  D  tcp 172.17.13.146 172.19.90.172
  34788  21
Dec 13 10:27:55  R1-RE0 /kernel: FW: fxp0.0  D igmp 172.19.91.95 224.0.0.1  0  0
. . .
Dec 13 10:34:41  R1-RE0 /kernel: FW: fxp0.0  D  udp 172.19.91.43 172.19.91.255
  138  138
Dec 13 10:34:55  R1-RE0 /kernel: FW: fxp0.0  D igmp 172.19.91.95 224.0.0.1  0  0
Dec 13 10:35:55  R1-RE0 /kernel: FW: fxp0.0  D igmp 172.19.91.95 224.0.0.1  0  0
Dec 13 10:36:55  R1-RE0 /kernel: FW: fxp0.0  D igmp 172.19.91.95 224.0.0.1  0  0
```

The result shown here is good. The only traffic not being accepted by other terms is coming from unauthorized hosts at 172.17.13.0/24, an address not included in the official lab topology, which shows the filter is having the desired effect. All the discarded traffic arrives on the shared OoB management network via fxp0, and appears to be a mix of IGMP, FTP, and NetBIOS. As a final confirmation, you confirm BGP and BFD session status at R1:

```
{master}[edit]
jnpr@R1-RE0# run show bgp summary
```

```
Groups: 3 Peers: 3 Down peers: 1
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...
10.3.255.2            65000       2010       2009       0       0    15:09:23 0/0/0/0
2001:db8:1::ff:2      65000        298        296       0       2     2:13:16 Establ
  inet6.0: 0/0/0/0
fd1e:63ba:e9dc:1::1   65010          0          0       0       0    17:52:23 Active
```

At this point, the EBGP session to the external BGP P1 device is expected to be down, but both the IPv6 and IPv4 IBGP sessions are established, as is the BFD session between R1 and R2. This BFD session is IPv4-based and runs over the ae0.1 interface to provide the IS-IS protocol with rapid fault detection capabilities:

```
{master}[edit]
jnpr@R1-RE0# show protocols isis
reference-bandwidth 100g;
level 1 disable;
interface ae0.1 {
    point-to-point;
    bfd-liveness-detection {
        minimum-interval 1000;
        multiplier 3;
    }
}
interface lo0.0 {
    passive;
}

{master}[edit]
jnpr@R1-RE0# run show bfd session
                                                  Detect   Transmit
Address                  State     Interface      Time     Interval  Multiplier
10.8.0.1                 Up        ae0.1          3.000     1.000        3

1 sessions, 1 clients
Cumulative transmit rate 1.0 pps, cumulative receive rate 1.0 pps
```

The continued operation of permitted services coupled with the lack of unexpected log entries from the discard-all action of both RE protection filters confirms they are working as designed and concludes the RE protection case study.

DDoS Protection Case Study

The MX Trio platforms began offering built-in DDoS protection starting with release v11.2. This feature makes use of the extensive host-bound traffic classification capabilities of the Trio chipset along with corresponding policers, implemented at various hierarchies within the system, to ensure the RE remains responsive in the event of excessive control plane exception traffic, such as can occur as the result of misconfigurations, excess scaling, or intentional DDoS types of attacks targeting a router's control plane.

The new low-level DDoS protection provides great benefit right out of the box, so to speak, but does not in itself mitigate the need for a RE protection filter to deny traffic that is not allowed or needed. When the new DDoS protection is combined with a strong RE filter, you can eliminate the need for policing functions in the filter, or for added protection you can continue to use RE filter-based policing as an added measure of safeguard, but in these cases you should ensure the RE filter-based policers have higher bandwidth values then the corresponding PFE and RE DDoS policers, or the policers in the RE will never have a chance to activate as the DDoS policers will see all the discard action. This is because a policer called from an input filter on the loopback interface is downloaded to the Trio PFE where it is executed before any DDoS policer functionality.

The Issue of Control Plane Depletion

As routers scale to provide service to more and more users with ever increasing numbers of services, it's not uncommon to find them operating near their capacity, especially in periods of heavy load such as route flap caused by network failures. With each new service comes additional load, but also the potential for unexpected resource usage either due to intent or in many cases because of buggy software or configuration errors that lead to unexpected operation.

Resource exhaustion can occur in a number of different places, each having their own set of operational issues. Run short on RIB/FIB and you may blackhole destinations or start using default routes with possibly undesirable paths. Low memory can lead to crashes, or slow reconvergence, as processes start swapping to disk. Run low on CPU, or on the internal communications paths needed to send and receive sessions to keep alive messages, and here comes even more trouble as BFD, BGP, and OSPF sessions begin flapping, which in turn only add more churn to an already too busy system.

In this section, the focus is on protecting the processing path, and therefore the control plane resources. Those control plane resources are needed to process remote access, routing protocols, and network management traffic as they make their way from a network interface through the PFE and onto the RE during periods of unexpected control plane traffic. The goal is to allow supported services, at reasonable levels, without allowing any one service or protocol to overrun all resources, a condition that can easily lead to denial of service for other protocols and users. Such a service outage can easily extend into the remote access needed to access a router in order to troubleshoot and correct the issue. There is little else in life as frustrating as knowing how to fix a problem, only to realize that because of the problem, you're unable to access the device to take corrective actions.

DDoS Operational Overview

The Juniper DDoS protection feature is based on two main components: the classification of host-bound control plane traffic and a hierarchical set of individual- and aggregate-level policers that cap the volume of control plane traffic that each protocol type is able to send to the RE for processing.

These policers are organized to match the hierarchical flow of protocol control traffic. Control traffic arriving from all ports of a line card converges at the card's Packet Forwarding Engine. Traffic from all PFEs converges into the line card/FPC. And lastly, control traffic from all line cards on the router converges on the routing engine. Similarly, the DDoS policers are placed hierarchically along the control paths so that excess packets are dropped as early as possible on the path. This design preserves system resources by removing excess malicious traffic so that the routing engine receives only the amount of traffic that it can actually process. In total, there can be as many as five levels of policing between ingress at the Trio PFE and processing at RE, and that's not counting any additional lo0-based filtering (with related policing) that can also be in effect.

In operation, control traffic is dropped when it violates a policer's rate limit. Each violation generates a notification in the syslog to alert the operator about a possible attack. Each violation is counted and its start time is noted, and the system also maintains a pointer to the last observed violation start and end times. When the traffic rate drops below the bandwidth violation threshold, a recovery timer determines when the traffic flow is considered to have returned to normal. If no further violation occurs before the timer expires, the violation state is cleared and a notification is again generated to report clearing of the DDoS event.

Once notified, it's the operator's responsibility to analyze the nature of the event to make a determination if the traffic type and volume that triggered the DDoS event was expected or abnormal. There is no easy answer here, as each network is scaled to different values with a differing mix of protocols and rate of churn. If the analysis concludes the volume of traffic was normal, then the related policers should be increased to avoid false alarms and potential service disruptions in the future. In contrast, protocols that are not used, or which are known to generate low message volume, can have their policers decreased.

The default policer settings are intentionally set high to ensure there are no unwanted side effects to preexisting installations as they are upgraded to newer code with DDoS protection support, which is enabled by default. In most cases, operators will want to characterize their network's expected control plane load and then *decrease* the default policer values to ensure they gain robust DDoS protection from the feature.

Policer states and statistics from each line card are relayed to the routing engine and aggregated. The policer states are maintained during a switchover. Note that during a GRES/NSR event, line card statistics and violation counts are preserved but RE policer statistics are not.

At this time, DDoS protection is a Trio-only feature. You can configure and commit it on a system that has older, DPC-style line cards but there will be no DDoS protection on those line cards. A chain is only as strong as the worst link; a system with a single line card that does not support DDoS is still vulnerable to an attack.

Host-Bound Traffic Classification

A modern multiservice router has to support a myriad of protocols, and multiprotocol support inherently assumes a method of recognizing each protocol so it can be directed to the correct processing daemon. The DDoS protection feature latches on to the Trio chipset's rich protocol classification capability to correctly recognize and bin a large number of subscriber access, routing, network management, and remote access protocols. The current list is already large and expected to grow:

```
{master}[edit system ddos-protection global]
jnpr@R1-RE0# run show ddos-protection version
DDOS protection, Version 1.0
  Total protocol groups       = 84
  Total tracked packet types  = 155
```

The display shows that in v1.0, there are 84 protocol groups with a total of 155 unique packets types that can be individually policed. The CLI's **?** feature is used to display the current list:

```
{master}[edit system ddos-protection]
jnpr@R1-RE0# set protocols ?
Possible completions:
> ancp                 Configure ANCP traffic
> ancpv6               Configure ANCPv6 traffic
+ apply-groups         Groups from which to inherit configuration data
+ apply-groups-except  Don't inherit configuration data from these groups
> arp                  Configure ARP traffic
> atm                  Configure ATM traffic
> bfd                  Configure BFD traffic
> bfdv6                Configure BFDv6 traffic
> bgp                  Configure BGP traffic
> bgpv6                Configure BGPv6 traffic
> demux-autosense      Configure demux autosense traffic
> dhcpv4               Configure DHCPv4 traffic
> dhcpv6               Configure DHCPv6 traffic
> diameter             Configure Diameter/Gx+ traffic
> dns                  Configure DNS traffic
> dtcp                 Configure dtcp traffic
> dynamic-vlan         Configure dynamic vlan exceptions
> egpv6                Configure EGPv6 traffic
> eoam                 Configure EOAM traffic
> esmc                 Configure ESMC traffic
> firewall-host        Configure packets via firewall 'send-to-host' action
> ftp                  Configure FTP traffic
> ftpv6                Configure FTPv6 traffic
> gre                  Configure GRE traffic
> icmp                 Configure ICMP traffic
> igmp                 Configure IGMP traffic
> igmp-snoop           Configure snooped igmp traffic
> igmpv4v6             Configure IGMPv4-v6 traffic
> igmpv6               Configure IGMPv6 traffic
> ip-fragments         Configure IP-Fragments
> ip-options           Configure ip options traffic
> ipv4-unclassified    Configure unclassified host-bound IPv4 traffic
> ipv6-unclassified    Configure unclassified host-bound IPv6 traffic
> isis                 Configure ISIS traffic
> jfm                  Configure JFM traffic
> l2tp                 Configure l2tp traffic
> lacp                 Configure LACP traffic
> ldp                  Configure LDP traffic
> ldpv6                Configure LDPv6 traffic
> lldp                 Configure LLDP traffic
> lmp                  Configure LMP traffic
> lmpv6                Configure LMPv6 traffic
> mac-host             Configure L2-MAC configured 'send-to-host'
> mlp                  Configure MLP traffic
> msdp                 Configure MSDP traffic
> msdpv6               Configure MSDPv6 traffic
> multicast-copy       Configure host copy due to multicast routing
> mvrp                 Configure MVRP traffic
> ntp                  Configure NTP traffic
> oam-lfm              Configure OAM-LFM traffic
> ospf                 Configure OSPF traffic
> ospfv3v6             Configure OSPFv3v6 traffic
> pfe-alive            Configure pfe alive traffic
> pim                  Configure PIM traffic
> pimv6                Configure PIMv6 traffic
> pmvrp                Configure PMVRP traffic
> pos                  Configure POS traffic
> ppp                  Configure PPP control traffic
> pppoe                Configure PPPoE control traffic
> ptp                  Configure PTP traffic
> pvstp                Configure PVSTP traffic
> radius               Configure Radius traffic
> redirect             Configure packets to trigger ICMP redirect
> reject               Configure packets via 'reject' action
> rip                  Configure RIP traffic
> ripv6                Configure RIPv6 traffic
> rsvp                 Configure RSVP traffic
> rsvpv6               Configure RSVPv6 traffic
> services             Configure services
> snmp                 Configure SNMP traffic
> snmpv6               Configure SNMPv6 traffic
> ssh                  Configure SSH traffic
> sshv6                Configure SSHv6 traffic
> stp                  Configure STP traffic
> tacacs               Configure TACACS traffic
> tcp-flags            Configure packets with tcp flags
> telnet               Configure telnet traffic
> telnetv6             Configure telnet-v6 traffic
> ttl                  Configure ttl traffic
> tunnel-fragment      Configure tunnel fragment
> virtual-chassis      Configure virtual chassis traffic
> vrrp                 Configure VRRP traffic
> vrrpv6               Configure VRRPv6 traffic
```

As extensive as the current protocol list is, it's just the outer surface of the MX router's protocol recognition capabilities; all of the protocol groups listed support aggregate-level policing and many also offer per-packet type policers that are based on the individual message types within that protocol. For example, the PPP over Ethernet (PPPoE) protocol group contains an aggregate policer in addition to numerous individual packet type policers:

```
{master}[edit system ddos-protection]
jnpr@R1-RE0# set protocols pppoe ?

Possible completions:
> aggregate            Configure aggregate for all PPPoE control traffic
+ apply-groups         Groups from which to inherit configuration data
+ apply-groups-except  Don't inherit configuration data from these groups
> padi                 Configure PPPoE PADI
> padm                 Configure PPPoE PADM
> padn                 Configure PPPoE PADN
> pado                 Configure PPPoE PADO
> padr                 Configure PPPoE PADR
> pads                 Configure PPPoE PADS
> padt                 Configure PPPoE PADT
{master}[edit system ddos-protection]
```

In contrast, ICMP is currently supported at the aggregate level only:

```
{master}[edit system ddos-protection protocols]
jnpr@R1-RE0# set icmp ?
Possible completions:
> aggregate            Configure aggregate for all ICMP traffic
+ apply-groups         Groups from which to inherit configuration data
+ apply-groups-except  Don't inherit configuration data from these groups
{master}[edit system ddos-protection protocols]
jnpr@R1-RE0# set icmp
```

Being able to recognize this rich variety of traffic at ingress means it can be directed to an equally rich set of policing functions to ensure the control plane load remains within acceptable limits. Given that many protocol groups support both individual packet type policers as well as aggregate-level policing at multiple locations in the host-bound processing path, the DDoS protection feature provides both effective and fine-grained control over host processing path resource protection.

A Gauntlet of Policers

Hierarchical policing is the DDoS prevention muscle behind the host-bound classification brains. This style of hierarchical policing is more akin to cascaded policers and should not be confused with the hierarchical policer discussed previously. The goal is to take action to limit excessive traffic as close to the source as possible, with each lower policer component feeding into a higher level policer, until a final policed aggregate for that protocol type is delivered to the RE for processing.

Figure 4-2 details the various DDoS policing hierarchies in the context of the PPPoE protocol group.

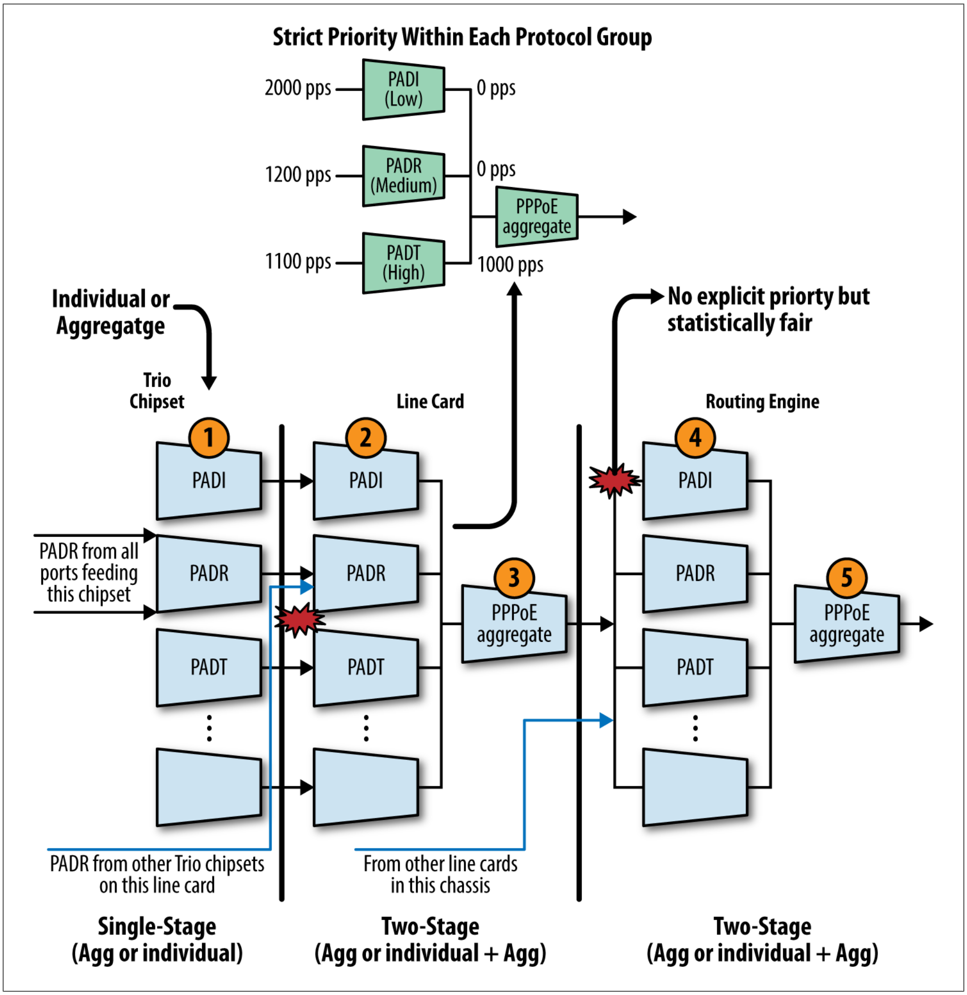

Strict Priority Within Each Protocol Group
2000 pps
PADI (Low)
0 pps
1200 pps
PADR (Medium)
0 pps
PPPoE aggregate
1100 pps
PADT (High)
1000 pps
Individual or Aggregatge
No explicit priorty but statistically fair
Trio Chipset
Line Card
Routing Engine
1
2
3
4
5
PADI
PADR
PADT
PADR from all ports feeding this chipset
PPPoE aggregate
PADR from other Trio chipsets on this line card
From other line cards in this chassis
Single-Stage (Agg or individual)
Two-Stage (Agg or individual + Agg)
Two-Stage (Agg or individual + Agg)

Figure 4-2. DDoS policing Points for the PPPoE Family.

The first level of policing is performed at ingress to the Trio chipset, shown in step 1, where each protocol group is subjected to a single policing stage that is *either* aggregate or individual packet type based.

> Currently, DHCP uses only an aggregate-level policer at the PFE stage, as is also the case at all stages for protocols that don't support individual packet type policing. At the PFE and RE hierarchies, DHCP for IPv4 and IPv6 is handled by two-stage policing based on individual message types, in addition to an aggregate rate for the group.

The next level of policing occurs in the line card (FPC) level, as the aggregate stream from all PFEs housed on the FPC contend for their place in the host processing queue. In most cases, including DHCP, the second line of policing consists of two stages: the first for individual message types and the second for the protocols group aggregate, which is shown at steps 2 and 3. Only those messages accepted at the first step are seen at stage 2, and any packet accepted at steps 1 and 2 is still very much subject to discard by the aggregate-level policer at step 3 when there's too much activity in its group.

Strict queuing is performed within individual message policers for a given protocol group to manage contention for the group's aggregate policer, based on a configured priority of high, medium, or low. The strict priority handling is shown at the top of the figure, where PADT traffic consumes all 1,000 PPS of the group's aggregate allowance even though other PPPoE message types are waiting. Here, PPPoE Active Discovery Termination (PADT) is considered more important than PPPoE Active Discovery Initiation (PADI), as it allows the release of PPPoE resources, which in turn facilitates the acceptance of new connections. Given the strict priority, all PADI will be dropped if PADT packets use up all the tokens of the PPPoE aggregate policer.

> Because high-priority traffic can starve lower priority traffic within its group, you should thoroughly consider modifying the priority for a given message type as the defaults have been carefully designed for optimal performance in a widest range of use cases.

The final level of policing hierarchy occurs within the RE itself, with another round of protocol group-based two-stage policing, shown in steps 4 and 5 within Figure 4-2. The output of this final stage consists of all the packets types for that group that were accepted by all policing stages in the path, which is then handed off to the associated daemon for message processing, assuming there are no lo0 filters or policers also in the host processing path.

The net result is a minimum of three policing stages for protocols that don't have individual packet type policers and five for those that do. Aggregate-only groups currently include ANCP, dynamic VLAN, FTP, and IGMP traffic. Groups that support both stages of policing currently include DHCPv4, MLP, PPP, PPPoE, and virtual chassis traffic. As the feature matures, groups that are currently aggregate level-only can be enhanced to support individual message type policing as the need arises.

By default, all three stages of policing (Trio chipset, line card, and routing engine) have the same bandwidth and burst limits for a given packet type. This design enables all the control traffic from a chipset and line card to reach the RE, as long as there is no competing traffic of the same type from other chipsets or line cards. When competing traffic is present, excess packets are dropped at the convergence points, which are the line card for all competing chipsets and the RE for all competing line cards. You can use a scaling factor to reduce the first two stages below the default values (100% of that used in the RE) to fine tune performance.

Note that there is no priority mechanism at the aggregate policer merge points, as shown in Figure 4-2. While there is no explicit prioritization, the bandwidth is allocated in a statistically fair manner, which is to say, higher rate traffic streams get proportionally more bandwidth than lower rate streams, and by the same token, during congestion higher rate streams will also see more discards.

> At the time of this chapter's writing, the CLI incorrectly offered priority as an option for aggregate policers. PR 722873 was raised to correct the issue.

The default policer values are intentionally set high to ensure valid services are not disrupted, given the DDoS feature is enabled by default, and each network varies with regard to what is considered a normal control plane load. Also, there is no one default size for all protocol groups because some message types are processed locally in the line card, and so can have a higher value, and the processing load can vary significantly for those that are sent to the RE. To gain maximum DDoS *prevention*, rather than *after-the-fact notification*, it's expected that each network operator will *reduce* policer values from their generous defaults *after analyzing actual load* in their network.

> Any time you lower a policer from its default, pay special attention to any alerts that may indicate it's too low for your network. Such a condition can lead to an unintentional local DDoS attack when the more aggressive policer begins discarding valid protocol traffic.

Configuration and Operational Verification

The DDoS prevention feature is configured at the `[edit system ddos-protection]` hierarchy. While there, you can alter the default policer and priority values for a long list of protocols, configure tracing, or modify global operating characteristics such as disabling RE or FPC level DDOS policers and event logging.

```
{master}[edit system ddos-protection]
jnpr@R1-RE0# set ?
```

```
Possible completions:
+ apply-groups         Groups from which to inherit configuration data
+ apply-groups-except  Don't inherit configuration data from these groups
> global               DDOS global configurations
> protocols            DDOS protocol parameters
> traceoptions         DDOS trace options
{master}[edit system ddos-protection]
```

Disabling and Tracing

You can disable policing at the FPC level (but not at the Trio PFE level) by including the `disable-fpc` statement. Likewise, you can use the `disable-routing-engine` statement to do the same for the RE's policers.

```
{master}[edit system ddos-protection global]
jnpr@R1-RE0# show
disable-routing-engine;
disable-fpc;
```

The two combined disable the last two levels of policing hierarchy, the FPC and RE levels; currently, the ingress Trio PFE-level policers cannot be disabled. Note that even when disabled the related daemon continues to run, and control plane policing *remains in effect* at the Trio PFE level. That last part no doubt sounds confusing, and so bears some clarification. Currently, you cannot disable the PFE level of policing, but the default values assigned to the policers are generally higher than that supported by the FPC-RE path, and so even though they remained enabled they are effectively transparent for traffic that needs to makes its way to the host.

> In the Junos v11.4 release, the CLI always shows the RE and FPC levels of policing as enabled, even when they have been globally disabled. PR 722873 was raised to track this issue.

If desired, you can completely disable the DDoS daemon, called `jddosd`, which collects policer statistics and generates logging of events, with a `system processes ddos-protection disable` configuration statement. If unexpected behavior is observed, and nothing else seems to help, consider restarting the process with a `restart ddos-protection` operational mode command.

In the initial release, the default DDoS policer values are equal to the same "higher than host path can support" rates as are used when the feature is disabled. This means the only real effect to disabling the feature when defaults are in place is whether or not you receive alerts when a policer is violated. This also means that if you do not model your network's control plane loads and reduce the default policer values accordingly, you are not gaining nearly as much protection from the DDoS feature as you could.

The decision to use default values that are higher than the host-bound path can actually support is based on the feature being enabled by default and the desire to be extra cautious about changing behavior when a customer upgrades to a newer version with DDoS support.

You can enable tracing to get additional information about DDoS operation and events by including trace flags—tracing is disabled by default. If desired, you can specify a log name and archive settings, rather than settle for the default `/var/log/ddosd` syslog, which by default is allowed to be 128 Kbytes before it's saved as one of three rolling archive files named `ddosd.0` through `ddosd.2`. The currently supported trace flags are displayed:

```
{master}[edit system ddos-protection]
jnpr@R1-RE0# set traceoptions flag ?
Possible completions:
  all                    Trace all areas of code
  config                 Trace configuration code
  events                 Trace event code
  gres                   Trace GRES code
  init                   Trace initialization code
  memory                 Trace memory management code
  protocol               Trace DDOS protocol processing code
  rtsock                 Trace routing socket code
  signal                 Trace signal handling code
  state                  Trace state handling code
  timer                  Trace timer code
  ui                     Trace user interface code
{master}[edit system ddos-protection]
jnpr@R1-RE0# set traceoptions flag
```

A typical trace configuration is shown, in this case creating a syslog called `ddos_trace` with a file size of 10 Mbytes, tracking events and protocol-level operations. DDoS logging occurs at the `notice` severity level, so if you specify something less severe (like `info`) you will not see any trace logs:

```
{master}[edit system ddos-protection]
jnpr@R1-RE0# show traceoptions
file ddos_trace size 10m;
level notice;
flag protocol;
flag events;
```

Granted, there is not much to see on a system that's not currently under some type of attack:

```
{master}[edit system ddos-protection]
jnpr@R1-RE0# run show log ddos_trace

{master}[edit system ddos-protection]
jnpr@R1-RE0#
```

Configure Protocol Group Properties

You can configure aggregate (and individual packet type) policing parameters when supported by the protocol group at the `[edit system ddos-protection protocols]` hierarchy. In most cases, a given group's aggregate policer has a larger bandwidth and burst setting, which is calculated on a per packet basis, than any individual packet type policer in the group; however, the sum of individual policers can exceed the group's aggregate rate. By default, the FPC- and Trio- PFE-level policers inherit bandwidth and burst size percentages values that are based on 100% of the aggregate or individual packet policer rate used at the RE level. From here, you can reduce or scale down the FPC percentages to limit them to a value below the RE policer rates, when desired. Again, the default setting of matching FPC to RE rate ensures that when no excess traffic is present, all messages accepted by the Trio policers are also accepted by the FPC-level policers, which in turn are also accepted by the RE-level policers.

In addition to policer parameters, you can also configure whether an individual policer type should bypass that group's aggregate policer (while still having its individual packet type statistics tracked), whether exceptions should be logged, the scheduling priority for individual packet type policers, and the recovery time. You can also disable RE or FPC level policing on a per protocol group/message type basis.

This example shows aggregate and individual packet type policer settings for the `ip-options` group:

```
protocols {
    ip-options {
        aggregate {
            bandwidth 10000;
            burst 500;
        }

        unclassified {
            priority medium;
        }

        router-alert {
            bandwidth 5000;
            recover-time 150;
            priority high;
        }
    }
}
```

The bandwidth and burst settings are measured in units of packets per second. The example shown explicitly sets the bandwidth and burst values for the ICMP aggregate policer and router alert individual message policers, and modifies the unclassified ICMP packet type to medium priority from its default of low. The router alert packet type has high priority by default; this example explicitly sets the default value. When burst size is not explicitly configured for an individual packet type, it inherits a value based on the aggregate's default using a proprietary mechanism that varies the burst size according to the assigned priority, where high-priority gets a higher burst size.

In this case, the aggregate rate has been reduced from 20K PPS to 10K PPS with a 500 packet burst size. The router alert individual message type has its bandwidth set to one-half that of the aggregate at 5K PPS; has been assigned a 150-second recovery time, which determines how long the traffic has to be below the threshold before the DDoS event is cleared; and has been assigned a high priority (which was the default for this message type). The only change made to the unclassified packet type is to assign it a medium priority. This change does not really buy anything for this specific protocol group example, because the `ip-option` group only has two members contending for the aggregate. After all, a medium priority setting only matters when there is another member using low, given the strict priority that's in effect when an individual packet type policer contends with other individual packet policers for access to the aggregate policer's bandwidth. The high priority router alert messages can starve the unclassified group just as easily, regardless of whether it uses a medium or low priority. Note that in this example starvation is not possible because the group's aggregate packet rate exceeds the individual rate allowed for IP optioned packets. Starvation will become an issue if the group's aggregate had been set to only 5K, so pay attention to priority settings in relation to the aggregate rate for a given protocol type.

Verify DDoS Operation

You now confirm the configured settings and expected operation using various forms of the `show ddos-protection` operational mode command:

```
{master}[edit]
jnpr@R1-RE0# run show ddos-protection ?
Possible completions:
  protocols            Show protocol information
  statistics           Show overall statistics
  version              Show version
{master}[edit]
jnpr@R1-RE0# run show ddos-protection
```

Most of the meat is obtained with the `protocols` switch, as demonstrated in the following. The version option displays info on DDoS version along with the numbers of classified protocols:

```
{master}[edit]
jnpr@R1-RE0# run show ddos-protection version
DDOS protection, Version 1.0
```

```
  Total protocol groups       = 84
  Total tracked packet types  = 155
```

The statistics option provides a quick summary of current DDoS state:

```
{master}[edit]
jnpr@R1-RE0# run show ddos-protection statistics
DDOS protection global statistics:
  Currently violated packet types:    0
  Packet types have seen violations:  0
  Total violation counts:             0
```

In this example, let's focus on the `ip-options` group and begin with the default parameters for this group:

```
{master}[edit]
jnpr@R1-RE0# run show ddos-protection protocols ip-options parameters brief
Number of policers modified: 0
Protocol    Packet      Bandwidth Burst  Priority Recover   Policer Bypass FPC
group       type        (pps)     (pkts)          time(sec) enabled aggr.  mod
ip-opt      aggregate   20000     20000  high     300       Yes     --     No
ip-opt      unclass..   10000     10000  low      300       Yes     No     No
ip-opt      rt-alert    20000     20000  high     300       Yes     No     No
```

The output confirms the group consists of an aggregate and two individual message types. The default values for bandwidth and burst are assigned, as are the individual priorities. You also see that neither individual message is allowed to bypass the aggregate and that the policers are enabled. The configuration is modified as per the previous example, and the changes are confirmed:

```
{master}[edit]
jnpr@R1-RE0# show | compare
[edit system]
+   ddos-protection {
+       traceoptions {
+           file ddos_trace size 10m;
+           level info;
+           flag protocol;
+           flag events;
+       }
+       protocols {
+           ip-options {
+               aggregate {
+                   bandwidth 10000;
+                   burst 500;
+               }
+               unclassified {
+                   priority medium;
+               }
+               router-alert {
+                   bandwidth 5000;
+                   recover-time 150;
+                   priority high;
+               }
+           }
+       }
+   }

{master}[edit]
jnpr@R1-RE0# run show ddos-protection protocols ip-options parameters brief
Number of policers modified: 3
Protocol    Packet      Bandwidth Burst  Priority Recover   Policer Bypass FPC
group       type        (pps)     (pkts)          time(sec) enabled aggr.  mod
ip-opt      aggregate   10000*    500*   high     300       Yes     --     No
ip-opt      unclass..   10000     10000  medium*  300       Yes     No     No
ip-opt      rt-alert    5000*     20000  high     150*      Yes     No     No
```

The output confirms the changes have taken effect; any nondefault value is called out with an "*." Note the default burst values have been calculated in relation to the relative priority factored against the burst aggregate's burst size, as was described previously. Use the show ddos-protection protocols command to display current violation state, traffic statistics, and details on the aggregate and individual packet type policer information for all or a selected protocol group:

```
{master}[edit system ddos-protection]
jnpr@R1-RE0# run show ddos-protection protocols ip-options ?
Possible completions:
  <[Enter]>              Execute this command
  |                      Pipe through a command
  parameters             Show IP-Options protocol parameters
  statistics             Show IP-Options statistics and states
  violations             Show IP-Options traffic violations
  aggregate              Show aggregate for all ip options traffic information
  unclassified           Show unclassified ip options traffic information
  router-alert           Show Router alert options traffic information
{master}[edit system ddos-protection]
jnpr@R1-RE0# run show ddos-protection protocols ip-options
```

The system baseline is now examined to confirm no current violations and that there has been very little ICMP activity since this system was booted:

```
{master}[edit]
jnpr@R1-RE0# run show ddos-protection protocols ip-options violations
Number of packet types that are being violated: 0

{master}[edit]
jnpr@R1-RE0# run show ddos-protection protocols ip-options statistics brief
Protocol    Packet      Received        Dropped        Rate     Violation State
group       type        (packets)       (packets)      (pps)    counts
ip-opt      aggregate   1               0              0        0         Ok
ip-opt      unclass..   0               0              0        0         Ok
ip-opt      rt-alert    1               0              0        0         Ok
```

Not only are the current traffic rate counters at 0, but the cumulative counter is also very low, with a single router alert IP optioned packet having been detected thus far. To see details, omit the brief switch:

```
{master}[edit]
jnpr@R1-RE0# run show ddos-protection protocols ip-options router-alert
Protocol Group: IP-Options
```

```
  Packet type: router-alert (Router alert options traffic)
    Individual policer configuration:
      Bandwidth:        5000 pps
      Burst:            20000 packets
      Priority:         high
      Recover time:     150 seconds
      Enabled:          Yes
      Bypass aggregate: No
    System-wide information:
      Bandwidth is never violated
      Received:  1                  Arrival rate:     0 pps
      Dropped:   0                  Max arrival rate: 0 pps
    Routing Engine information:
      Policer is never violated
      Received:  1                  Arrival rate:     0 pps
      Dropped:   0                  Max arrival rate: 0 pps
        Dropped by aggregate policer: 0
    FPC slot 1 information:
      Bandwidth: 100% (5000 pps), Burst: 100% (20000 packets), enabled
      Policer is never violated
      Received:  0                  Arrival rate:     0 pps
      Dropped:   0                  Max arrival rate: 0 pps
        Dropped by aggregate policer: 0
    FPC slot 2 information:
      Bandwidth: 100% (5000 pps), Burst: 100% (20000 packets), enabled
      Policer is never violated
      Received:  1                  Arrival rate:     0 pps
      Dropped:   0                  Max arrival rate: 0 pps
        Dropped by aggregate policer: 0
```

The output for the router alert individual packet policer confirms systemwide settings, as well as traffic and policer statistics for both the RE and FPC hierarchies. Note that the first-stage Trio PFE-level stats are not displayed in the CLI, but violations are reported via the FPC housing that Trio PFE. With the information provided, you can quickly discern if there is currently excess router alert traffic, whether excess traffic has been detected in the past, and if so, the last violation start and end time. The per FPC displays include any alerts or violation that have been detected at either the Trio chipset or the FPC policing levels, information that allows you to quickly determine the ingress points for anomalous control plane traffic.

You cannot clear violation history except with a system reboot. You can clear a specific group's statistics or clear a current violation state using the `clear ddos-protection protocols` command:

```
{master}[edit]
jnpr@R1-RE0# run clear ddos-protection protocols ip-options ?
Possible completions:
  statistics           Clear IP-Options statistics
  states               Reset IP-Options states
  aggregate            Clear aggregate for all ip options traffic information
  unclassified         Clear unclassified ip options traffic information
  router-alert         Clear Router alert options traffic information
```

Late Breaking DDoS Updates

As noted previously, the DDoS feature is new. Like most new things, it continues to evolve based on customer and field engineer feedback. The DDoS coverage in this section was based on the v11.4R1.9 Junos release. As it happens, the v11.4R2 release contained updates that enhance the feature, and this seemed a good place to capture them. As always, you should consult the latest Junos feature documentation for your release to ensure you stay abreast of feature evolution. User visible changes to DDoS in 11.4R2 include:

- It's now possible to include the `disable-fpc` statement at the `[edit system ddos-protection protocols protocol-group (aggregate | packet-type)]` hierarchy level to disable policers on all line cards for a particular packet type or aggregate within a protocol group. The ability to configure this statement globally or for a particular line card remains unchanged.
- The `show ddos-protection protocols` command now displays `Partial` in the Enabled field to indicate when some of the instances of the policer are disabled, and displays `disabled` when all policers are disabled.
- The routing engine information section of the `show ddos-protection protocols` command output now includes fields for bandwidth, burst, and state.
- The `show ddos-protection protocols parameters` command and the `show ddos-protection protocols statistics` command now include a `terse` option to display information only for active protocol groups—that is, groups that show traffic in the Received (packets) column. The `show ddos-protection protocols parameters` command also displays `partial` for policers that have some disabled instances.

DDoS Case Study

This case study is designed to show the DDoS prevention feature in action. It begins with the modified configuration for the `ip-options` group discussed in the previous section. So far, no DDoS alerts or trace activity have been detected on R1, as evidenced by the lack of alerts in the system log files:

```
{master}

{master}
jnpr@R1-RE0> show log messages | match ddos

{master}
jnpr@R1-RE0> show log ddos_trace

{master}
jnpr@R1-RE0>
```

No real surprise, given the system's lab setting and the lack of hostile intent in those who, having had the pleasure of using it, have developed somewhat affectionate feelings for the little chassis. At any extent, in the interest of moving things along, the author has agreed to use a router tester to target R1 with the proverbial boatload of IP optioned packets. After all, DDoS protection is in place so no routers should be harmed in the experiment. In this case, all the packets are coded with the infamous router alert—recall this option forces RE-level processing and thereby serves as a potential attack vector among the more shady characters that share our civilization.

Figure 4-3 shows the topology details for the DDoS protection lab.

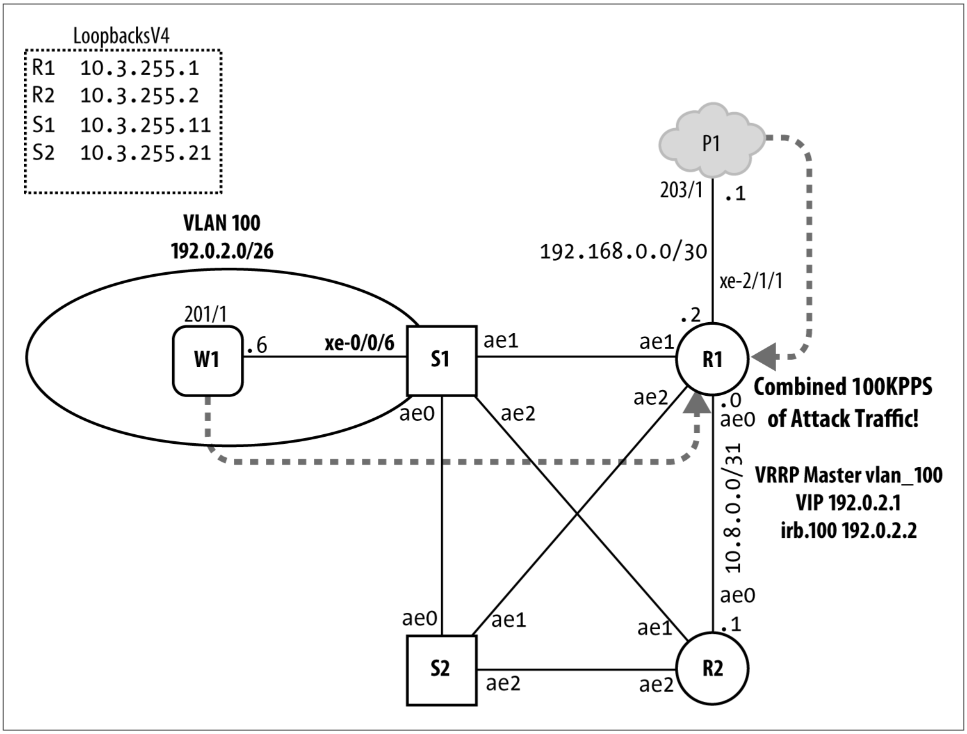


Figure 4-3. DDoS Protection Lab Topology.

The plan is to generate two identical streams of these black-hat- wearing packets, one via the xe-0/0/6 Layer 2 access interface at S1 and the other over the xe-2/1/1 Layer 3 interface connecting R1 to P1. Both packet streams are sent from IP address 192.0.2.20 and destined to the 192.0.2.3 address assigned to r2's VLAN 100 IRB interface. Recall that the presence of the `router-alert` option forces R1 to examine this transit traffic even though it's not addressed to one of its local IP addresses. The Ethernet frame's destination MAC address is set to all 1's broadcast, a setting that ensures copies of the same stream will be accepted for processing/routing on R1's Layer 3 interface while also flooded in the VLAN 100 Layer 2 domain by S1. The packets are 128 bytes long at Layer 2 and are generated at 50,000 packets per second, for a combined load of 100K PPS.

That is a fair amount of traffic for an RE to process, each and every second. This could be dangerous, if not for DDoS protection!

The Attack Has Begun!

The stage is set, and the DDoS daemon is standing by, ready to take the best of whatever shot you can throw, so traffic generation is initiated.

Oh, the humanity . . . But wait, the router is still responsive, there is no meltdown. In fact, the only real indication that anything is amiss is the syslog entry from `jddosd` reporting the violation:

```
jnpr@R1-RE0> show log messages | match ddos
Dec 19 18:16:56  R1-RE0 jddosd[1541]: DDOS_PROTOCOL_VIOLATION_SET: Protocol
  IP-Options:router-alert is violated at fpc 2 for 1 times, started at
  2011-12-19 18:16:56 PST, last seen at 2011-12-19 18:16:56 PST
```

The syslog information flags you as to the nature of the attack traffic, as well as the affected FPC, in this case FPC 2. The same information is also found to be written to the DDoS trace file, which can be handy if someone has disabled DDoS logging globally, as the global disable logging statement only controls logging to the main syslog messages file, not to a DDoS-specific trace file:

```
{master}[edit]
jnpr@R1-RE0# run show log ddos_trace

Dec 19 18:16:56 Protocol IP-Options:router-alert is violated at fpc 2 for 1 times,
  started at 2011-12-19 18:16:56 PST, last seen at 2011-12-19 18:16:56  PST
```

Analyze the Nature of the DDoS Threat

Once you are alerted that abnormal levels of control plane traffic has been detected, you can quickly narrow down the nature and scope of the anomaly using the following process.

First, confirm violation state with a `show ddos-protection statistics` command:

```
{master}
jnpr@R1-RE0> show ddos-protection statistics
DDOS protection global statistics:
  Currently violated packet types:    1
  Packet types have seen violations:  1
  Total violation counts:             1
```

To display the scope of protocols currently involved, add the `violations` keyword:

```
{master}
jnpr@R1-RE0> show ddos-protection protocols violations
Number of packet types that are being violated: 1
Protocol    Packet      Bandwidth  Arrival    Peak       Policer bandwidth
group       type        (pps)      rate(pps)  rate(pps)  violation detected at
ip-opt      rt-alert    5000       100004     100054     2011-12-19 18:16:56 PST
          Detected on: FPC-2
```

With no other protocols in a violation state, and knowing it's not just IP options but specifically router alerts that make up the attack, move on to display the details for that traffic type:

```
{master}
jnpr@R1-RE0> show ddos-protection protocols ip-options router-alert
Protocol Group: IP-Options

  Packet type: router-alert (Router alert options traffic)
    Individual policer configuration:
      Bandwidth:        5000 pps
      Burst:            20000 packets
      Priority:         high
      Recover time:     150 seconds
      Enabled:          Yes
      Bypass aggregate: No
    System-wide information:
      Bandwidth is being violated!
        No. of FPCs currently receiving excess traffic: 1
        No. of FPCs that have received excess traffic:  1
        Violation first detected at: 2011-12-19 18:16:56 PST
        Violation last seen at:      2011-12-19 18:19:33 PST
        Duration of violation: 00:02:37 Number of violations: 1
      Received:  15927672           Arrival rate:     100024 pps
      Dropped:   10402161           Max arrival rate: 100054 pps
    Routing Engine information:
      Policer is never violated
      Received:  374395             Arrival rate:     2331 pps
      Dropped:   0                  Max arrival rate: 2388 pps
        Dropped by aggregate policer: 0
    FPC slot 1 information:
      Bandwidth: 100% (5000 pps), Burst: 100% (20000 packets), enabled
      Policer is never violated
      Received:  0                  Arrival rate:     0 pps
      Dropped:   0                  Max arrival rate: 0 pps
        Dropped by aggregate policer: 0
    FPC slot 2 information:
      Bandwidth: 100% (5000 pps), Burst: 100% (20000 packets), enabled
      Policer is currently being violated!
        Violation first detected at: 2011-12-19 18:16:56 PST
        Violation last seen at:      2011-12-19 18:19:33 PST
        Duration of violation: 00:02:37 Number of violations: 1
      Received:  15927672           Arrival rate:     100024 pps
      Dropped:   10402161           Max arrival rate: 100054 pps
        Dropped by this policer: 10402161
        Dropped by aggregate policer: 0
```

The very fact that this output is obtained proves that R1 has remained responsive throughout the event, and thereby the effectiveness of the new Trio DDoS protection. Note how the stats for policing at the RE level show a peak load of only 2,388 PPS, while the FPC 2 statistics confirm an arrival rate of 100,000 PPS. And large numbers of drops are confirmed, which accounts for the difference in Trio/FPC policing load versus the volume of traffic that is actually making it to the RE.

The display also confirms that all of the bad traffic ingresses on FPC2. Just knowing that can help you apply filters or other methods to begin tracking back to the point at which the bad traffic ingresses your network, so you can either disable the peering interface or apply a filter to block the traffic before it endangers your network.

It's often helpful to display protocol group-level information, which also includes any individual packet policers, even when you know a specific violation is caught with an individual packet policer, such as the case with the router alert example being discussed. The group-level displays combined information from all five policing points, albeit in what can be a rather long display, which helps you identify Trio-level PFE policing actions from those in the FPC or RE. To best illustrate how the DDoS hierarchical policers work, the statistics and state from the last experiment are cleared:

```
{master}
jnpr@R1-RE0> clear ddos-protection protocols statistics
jnpr@R1-RE0> clear ddos-protection protocols state
```

And the traffic generator is altered to send one million router alert packets, at a 100K PPS rate, over a single interface. The round numbers should help make later analysis that much easier. After the traffic is sent, the protocol group-level DDoS policer information is displayed:

```
{master}
jnpr@R1-RE0> show ddos-protection protocols ip-options
Protocol Group: IP-Options

  Packet type: aggregate (Aggregate for all options traffic)
    Aggregate policer configuration:
      Bandwidth:        10000 pps
      Burst:            500 packets
      Priority:         high
      Recover time:     300 seconds
      Enabled:          Yes
    System-wide information:
      Aggregate bandwidth is never violated
      Received:  71751               Arrival rate:     0 pps
      Dropped:   0                   Max arrival rate: 6894 pps
    Routing Engine information:
      Aggregate policer is never violated
      Received:  40248               Arrival rate:     0 pps
      Dropped:   0                   Max arrival rate: 4262 pps
        Dropped by individual policers: 0
    FPC slot 1 information:
      Bandwidth: 100% (10000 pps), Burst: 100% (500 packets), enabled
      Aggregate policer is never violated
      Received:  0                   Arrival rate:     0 pps
      Dropped:   0                   Max arrival rate: 0 pps
        Dropped by individual policers: 0
    FPC slot 2 information:
      Bandwidth: 100% (10000 pps), Burst: 100% (500 packets), enabled
      Aggregate policer is never violated
      Received:  71751               Arrival rate:     0 pps
      Dropped:   31743               Max arrival rate: 6894 pps
        Dropped by individual policers: 31743

  Packet type: unclassified (Unclassified options traffic)
    Individual policer configuration:
      Bandwidth:        10000 pps
      Burst:            10000 packets
      Priority:         medium
      Recover time:     300 seconds
      Enabled:          Yes
      Bypass aggregate: No
    System-wide information:
      Bandwidth is never violated
      Received:  0                   Arrival rate:     0 pps
      Dropped:   0                   Max arrival rate: 0 pps
    Routing Engine information:
      Policer is never violated
      Received:  0                   Arrival rate:     0 pps
      Dropped:   0                   Max arrival rate: 0 pps
        Dropped by aggregate policer: 0
    FPC slot 1 information:
      Bandwidth: 100% (10000 pps), Burst: 100% (10000 packets), enabled
      Policer is never violated
      Received:  0                   Arrival rate:     0 pps
      Dropped:   0                   Max arrival rate: 0 pps
        Dropped by aggregate policer: 0
    FPC slot 2 information:
      Bandwidth: 100% (10000 pps), Burst: 100% (10000 packets), enabled
      Policer is never violated
      Received:  0                   Arrival rate:     0 pps
      Dropped:   0                   Max arrival rate: 0 pps
        Dropped by aggregate policer: 0

  Packet type: router-alert (Router alert options traffic)
    Individual policer configuration:
      Bandwidth:        5000 pps
      Burst:            20000 packets
      Priority:         high
      Recover time:     150 seconds
      Enabled:          Yes
      Bypass aggregate: No
    System-wide information:
      Bandwidth is being violated!
        No. of FPCs currently receiving excess traffic: 1
        No. of FPCs that have received excess traffic:  1
        Violation first detected at: 2011-12-19 19:00:43 PST
        Violation last seen at:      2011-12-19 19:00:53 PST
        Duration of violation: 00:00:10 Number of violations: 2
      Received:  1000000             Arrival rate:     0 pps
      Dropped:   819878              Max arrival rate: 100039 pps
    Routing Engine information:
      Policer is never violated
      Received:  40248               Arrival rate:     0 pps
      Dropped:   0                   Max arrival rate: 4262 pps
        Dropped by aggregate policer: 0
    FPC slot 1 information:
      Bandwidth: 100% (5000 pps), Burst: 100% (20000 packets), enabled
      Policer is never violated
      Received:  0                   Arrival rate:     0 pps
      Dropped:   0                   Max arrival rate: 0 pps
        Dropped by aggregate policer: 0
    FPC slot 2 information:
      Bandwidth: 100% (5000 pps), Burst: 100% (20000 packets), enabled
      Policer is currently being violated!
        Violation first detected at: 2011-12-19 19:00:43 PST
        Violation last seen at:      2011-12-19 19:00:53 PST
        Duration of violation: 00:00:10 Number of violations: 2
      Received:  1000000             Arrival rate:     0 pps
      Dropped:   819878              Max arrival rate: 100039 pps
        Dropped by this policer: 819878
        Dropped by aggregate policer: 0
```

There is a lot of information here; refer back to Figure 4-2 for a reminder on the five levels of DDoS policing that are possible, and let's take it one step at a time.

The first stage of DDoS policing occurs at the Trio FPC level. The ingress Trio statistics are at the bottom of the display, under the `Packet type: router-alert (Router alert options traffic)` heading. The maximum arrival rate of 1,000,039 PPS corresponds nicely with the traffic's burst length and rate parameters, as does the received count of 1,000,000 packets. The display confirms that this policer is currently violated, and, importantly, shows that 819,878 packets have been dropped.

Recall that the goal of Junos DDoS protection is to first recognize when there is excessive control plane traffic and then to cut it off as close to the source and as far away from the RE as possible. The numbers confirm that over 80% of the excess traffic was discarded, and this in the first of as many as five policing stages. Clearly, DDoS has acted to preserve control plane resources farther up the line. With the drops shown at this stage, there should be some 180,122 options packets still making their way up north to the land of the RE.

The next step is the FPC policer, which for this group is a two-stage policer with individual- and aggregate-level policing. Its details are in the `FPC slot 2 information` under the `aggregate (Aggregate for all options traffic)` heading. Here is where you have to do some detective work. The display confirms that the FPC-level aggregate policer was never violated, but at the same time it shows 31,743 drops, which therefore had to come from its individual packet policer stage. The display also shows the FPC policing stage received only 71,751 packets, which is well short of the 180,122 that made it through the Trio PFE-level policer.

When asked about the discrepancy between DDoS stages, a software engineer confirmed the presence of "legacy policing functions that may also drop excess traffic on the host path." For example, the built-in ICMP rate limiting function that is viewed with a `show system statistics icmp` command. The defaults can be altered via the `set system internet-options` configuration statement:

```
[edit]
jnpr@R4# set system internet-options icmpv?
Possible completions:
> icmpv4-rate-limit    Rate-limiting parameters for ICMPv4 messages
> icmpv6-rate-limit    Rate-limiting parameters for ICMPv6 messages
```

Here, the V4 options are shown:

```
[edit]
jnpr@R4# set system internet-options icmpv4-rate-limit ?
Possible completions:
  bucket-size          ICMP rate-limiting maximum bucket size (seconds)
  packet-rate          ICMP rate-limiting packets earned per second
[edit]
jnpr@R4# set system internet-options icmpv4-rate-limit
```

The moral of the story is you should not expect 100% correlation of the counters shown at the various policing stages, as this data only reflects actions associated with DDoS processing and not those of other host protection mechanisms that may continue to coexist. Recall the goal of the feature is to protect the RE while providing the operator with the information needed to ascertain the scope and nature of an attack, not to provide statistics suitable for usage-based billing.

The fact that the FPC's aggregate policer was never violated is a testament to the effectiveness of the actions at the first stage. With the FPC showing receipt of 71,757 packets, and factoring the 31,743 discards, there should be about 40,014 packets left to make their way through the final policing stage in the RE itself.

The RE's policer stats are shown in a few places. Looking at the one under the group aggregate, it's possible to see it has received a total of 40,248 packets. The display also confirms no discards in the RE policer at either the individual or aggregate levels. The number is slightly higher than the 40,014 that were assumed to have left the FPC, perhaps due to some other legacy system function, but the numbers still mesh relatively well with the known nature of this attack. In the end, the fact that 1M of these puppies were sent while the RE only had to deal with 40K of them, all due to a hardware-based feature that has no forwarding performance impact, should really drive home the benefits of this Trio-only feature.

Mitigate DDoS Attacks

Once you have analyzed the nature of a DDoS violation, you will know what type of traffic is involved and on which PFEs and line cards/FPCs the traffic is arriving. Armed with this information, you can manually begin deployment of stateless filters on the upstream nodes until you reach the border of your network where you can disable the offending peer or apply a filter to discard or rate limit the offending traffic as close to its source as possible. Once the fire is out, so to speak, you can contact the administrators of the peering network to obtain their assistance in tracing the attack to the actual sources of the attack, where corrective actions can be taken.

BGP Flow-Specification to the Rescue

The Junos BGP flow-specification (flow-spec or flow route) feature uses MP-BGP to rapidly deploy filter and policing functionality among BGP speaking nodes on both an intra- and inter-Autonomous System basis. This feature that is well suited to mitigating the effects of a DDoS attack, both locally and potentially over the global Internet, once the nature of the threat is understood. A flow specification is an n-tuple filter definition consisting of various IPv4 match criteria and a set of related actions that is distributed via MP-BGP so that remote BGP speakers can dynamically install stateless filtering and policing or offload traffic to another device for further handling. A given IP packet is said to match the defined flow if it matches all the specified criteria. Flow routes are an aggregation of match conditions and resulting actions for matching traffic that include filtering, rate limiting, sampling, and community attribute modification. Using flow-spec, you can define a filter once, and then distribute that filter throughout local and remote networks–just the medicine needed to nip a DDoS attack as close to the bud as possible.

Unlike BGP-based Remote Triggered Black Holes (RTBH), flow-spec gives you the ability to match on a wide range of match criteria, rather than policy-based matches that are destination IP address–based. And again, with flow-spec you can define match and filtering conditions in a central local and then use BGP to push that information out to both internal and external BGP speakers.

BGP flow-specification network-layer reachability information (NLRI) and its related operation are defined in RFC 5575 "Dissemination of Flow Specification Rules." Note that Juniper publications still refer to the previous Internet draft "draft-ietf-idr-flow-spec-09," which relates to the same functionality. In operation, a flow-spec's filter match criteria are encoded within the flow-spec NLRI, whereas the related actions are encoded in extended communities. Different NLRI are specified for IPv4 versus Layer 3 VPN IPv4 to accommodate the added route-distinguisher and route targets. Once again, the venerable warhorse that is BGP shows its adaptability. A new service is enabled through opaque extensions to BGP, which allows operators to leverage a well-understood and proven protocol to provide new services or enhanced security and network robustness, as is the case with flow-spec. Flow-spec information is said to be opaque to BGP because its not BGP's job to parse or interpret the flow-spec payload. Instead, the flow-spec information is passed through the flow-spec validation module, and when accepted, is handed to the firewall daemon (`dfwd`) for installation into the PFEs as a stateless filter and/or policer.

In v11.4, Junos supports flow-spec NLRI for both main instance IPv4 unicast and Layer 3 VPN IPv4 unicast traffic. To enable flow-specification NLRI for main instance MP-BGP, you include the `flow` statement for the `inet` address family at the `[edit protocols bgp group group-name family inet]` hierarchy. To enable flow-specification NLRI for the `inet-vpn` address family, include the `flow` statement at the `[edit protocols bgp group group-name family inet-vpn]` hierarchy level. Note that the `flow` family is valid only for main instance MP-BGP sessions; you cannot use this family for BGP session within a VRF.

Local and received flow routes that pass validation are installed into the flow routing table `instance-name.inetflow.0`, where matching packets are then subjected to the related flow-spec's actions. Flow routes that do not pass validation are hidden in the related table null preference. Any change in validation status results in immediate update to the flow route. Received Layer 3 VPN flow routes are stored in the `bgp.invpnflow.0` routing table and still contain their Route Distinguishers (RD). Secondary routes are imported to one or more specific VRF tables according to vrf-import policies. Unlike the `inet` flow NLRI, `inet-vpn` flow routes are not automatically validated against a specific VRF's unicast routing information; this is because such an operation must be performed within a specific VRF context, and based on route-target the same flow NLRI can be imported into multiple VRFs.

Configure Local Flow-Spec Routes

You configure a local flow-specification for injection into BGP at the `routing-options` hierarchy, either in the main instance or under a supported instance type (VRF or VR). While some form of IDS may be used to provide alerts as to the need for flow-spec, in many cases operators will use SNMP alarms, RE protection filters, or the new DDoS feature to provide notification of abnormal traffic volumes. Using the details provided by these features allows the operator to craft one or more flow-specs to match on the attack vector and either filter outright or rate limit as deemed appropriate.

Flow-spec syntax is very much like a stateless filter; the primary difference is lack of a term function, as flow-specs consist of a single term. Otherwise, just like a filter, the flow-spec consists of a set of match criteria and related actions. As before, a match is only declared when all criteria in the `from` statement are true, else processing moves to the next flow-specification. The options for flow definition are displayed:

```
{master}[edit routing-options flow]
jnpr@R1-RE0# set ?
Possible completions:
+ apply-groups         Groups from which to inherit configuration data
+ apply-groups-except  Don't inherit configuration data from these groups
> route                Flow route
  term-order           Term evaluation order for flow routes
> validation           Flow route validation options
{master}[edit routing-options flow]
jnpr@R1-RE0# set
```

The `term-order` keyword is used to select between version 6 and later versions of the flow-spec specification, as described in the next section. Validation of flow routes, a process intended to prevent unwanted disruption from a feature that is intended to minimize disruption, is an important concept. It too is detailed in a following section. Currently, the `validation` keyword at the `[edit routing-options flow]` hierarchy is used to configure tracing for the validation process.

Supported match criteria include:

```
{master}[edit routing-options flow]
jnpr@R1-RE0# set route test ?
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
> match                Flow definition
> then                 Actions to take for this flow
  |                    Pipe through a command
{master}[edit routing-options flow]
jnpr@R1-RE0# set route test match ?
Possible completions:
+ apply-groups         Groups from which to inherit configuration data
+ apply-groups-except  Don't inherit configuration data from these groups
  destination          Destination prefix for this traffic flow
+ destination-port     Destination TCP/UDP port
+ dscp                 Differentiated Services (DiffServ) code point (DSCP)
+ fragment
+ icmp-code            ICMP message code
+ icmp-type            ICMP message type
+ packet-length        Packet length
+ port                 Source or destination TCP/UDP port
+ protocol             IP protocol value
  source               Source prefix for this traffic flow
+ source-port          Source TCP/UDP port
+ tcp-flags            TCP flags
{master}[edit routing-options flow]
jnpr@R1-RE0# set route test match
```

And the supported actions:

```
{master}[edit routing-options flow]
jnpr@R1-RE0# set route test then ?
Possible completions:
  accept               Allow traffic through
+ apply-groups         Groups from which to inherit configuration data
+ apply-groups-except  Don't inherit configuration data from these groups
  community            Name of BGP community
  discard              Discard all traffic for this flow
  next-term            Continue the filter evaluation after matching this flow
  rate-limit         Rate at which to limit traffic for this flow (9600..1000000000000)
  routing-instance     Redirect to instance identified via Route Target community
  sample               Sample traffic that matches this flow
{master}[edit routing-options flow]
jnpr@R1-RE0# set route test then
```

A sample flow-specification is shown:

```
{master}[edit routing-options flow]
jnpr@R1-RE0# show
route flow_http_bad_source {
    match {
        source 10.0.69.0/25;
        protocol tcp;
        port http;
    }
    then {
        rate-limit 10k;
        sample;
    }
}
```

After the filter chapter, the purpose of the `flow_http_bad_source` flow-specification should be clear. Once sent to a remote peer, you can expect matching HTTP traffic to be rate limited and sampled (according to its sampling parameters, which are not shown here).

Flow-Spec Algorithm Version. With BGP flow-spec, it's possible that more than one rule may match a particular traffic flow. In these cases, it's necessary to define the order at which rules get matched and applied to a particular traffic flow in such a way that the final ordering must not depend on the arrival order of the flow-specification's rules and must be constant in the network to ensure predictable operation among all nodes.

Junos defaults to the term-ordering algorithm defined in version 6 of the BGP flow-specification draft. In Junos OS Release v10.0 and later, you can configure the router to comply with the term-ordering algorithm first defined in version 7 of the BGP flow specification and supported through RFC 5575, "Dissemination of Flow Specification Routes." The current best practice is to configure the version 7 term-ordering algorithm. In addition, it's recommended that the same term-ordering version be used on all routing instances configured on a given router.

In the default term ordering algorithm (draft Version 6), a term with less specific matching conditions is always evaluated before a term with more specific matching conditions. This causes the term with more specific matching conditions to never be evaluated. Draft version 7 made a revision to the algorithm so that the more specific matching conditions are evaluated before the less specific matching conditions. For backward compatibility, the default behavior is not altered in Junos, even though the newer algorithm is considered better. To use the newer algorithm, include the `term-order standard` statement in the configuration.

Validating Flow Routes

Junos installs flow routes into the flow routing table only if they have been validated using the validation procedure described in the draft-ietf-idr-flow-spec-09.txt, *Dissemination of Flow Specification Rules*. The validation process ensures the related flow-spec NLRI is valid and goes on to prevent inadvertent DDoS filtering actions by ensuring that a flow-specification for a given route is only accepted when it is sourced from the same speaker that is the current selected active next-hop for that route. Specifically, a flow-specification NLRI is considered feasible if and only if:

- The originator of the flow-specification matches the originator of the best-match unicast route for the destination prefix embedded in the flow-specification.

- There are no more specific unicast routes, when compared with the flow destination prefix, that have been received from a different neighboring AS than the best-match unicast route, which has been determined in the first step.

The underlying concept is that the neighboring AS that advertises the best unicast route for a destination is allowed to advertise flow-spec information for that destination prefix. Stated differently, dynamic filtering information is validated against unicast routing information, such that a flow-spec filter is accepted if, and only if, it has been advertised by the unicast next-hop that is advertising the destination prefix and there are no unicast routes more specific than the flow destination, with a different next-hop AS number. Ensuring that another neighboring AS has not advertised a more specific unicast route before validating a received flow-specification ensures that a filtering rule affects traffic that is only flowing to the source of the flow-spec and prevents inadvertent filtering actions that could otherwise occur. The concept is that, if a given routing peer is the unicast next-hop for a prefix, then the system can safely accept from the same peer a more specific filtering rule that belongs to that aggregate.

You can bypass the validation process and use your own import policy to decide what flow-spec routes to accept using the `no-validate` switch:

```
[protocols bgp group <name>]
family inet {
    flow {
    no-validate <policy-name>;
    }
}
```

Bypassing the normal validation steps can be useful in the case where there is one system in the AS in charge of advertising filtering rules that are derived locally, perhaps via an Intrusion Detection System (IDS). In this case, the user can configure a policy that, for example, accepts BGP routes with an empty as-path to bypass the normal validation steps.

In addition, you can control the import and export of flow routes through import and export policy statements, respectively, which are applied to the related BGP peering sessions in conventional fashion. These policies can match on various criteria to include `route-filter` statements to match against the destination address of a flow route and the ability to use a `from rib inetflow.0` statement to ensure that only flow-spec routes can be matched. You can apply Forwarding Table export policy to restrict flow route export to the PFE. The default policy is to advertise all locally defined flow-routes and to accept for validation all received flow-routes.

You can confirm the validation status of a flow route with a `show route detail` command. In this, a Layer 3 VPN flow route is shown:

```
. . .
vrf-a.inetflow.0: 2 destinations, 2 routes (2 active, 0 holddown, 0 hidden)
10.0.1/24,*,proto=6,port=80/88 (1 entry, 1 announced)
        *BGP    Preference: 170/-101
                Next-hop reference count: 2
                State: <Active Ext>
                Peer AS: 65002
                Age: 3:13:32
                Task: BGP_65002.192.168.224.221+1401
                Announcement bits (1): 1-BGP.0.0.0.0+179
                AS path: 65002 I
                Communities: traffic-rate:0:0
                Validation state: Accept, Originator: 192.168.224.221
                Via: 10.0.0.0/16, Active
                Localpref: 100
                Router ID: 201.0.0.6
```

In the output, the `Validation state` field confirms the flow route was validated (as opposed to rejected) and confirms the originator or the flow route as IP address 192.168.224.221. The `via:` field indicates which unicast route validated the flow-spec route, which in this case was 10.0/16. Use the `show route flow validation` command to display information about unicast routes that are used to validate flow specification routes.

You can trace the flow-spec validation process by adding the `validation` flag at the `[edit routing-options flow]` hierarchy:

```
{master}[edit routing-options flow]
jnpr@R1-RE0# show
validation {
    traceoptions {
        file flow_trace size 10m;
        flag all detail;
    }
}
```

Limit Flow-Spec Resource Usage. Flow-spec routes are essentially firewall filters, and like any filter there is some resource consumption and processing burden that can vary as a function of the filter's complexity. However, unlike a conventional filter that requires local definition, once flow-spec is enabled on a BGP session, the remote peer is effectively able to cause local filter instantiation, potentially up until the point of local resource exhaustion, which can lead to bad things. To help guard against excessive resource usage in the event of misconfigurations or malicious intent, Junos allows you to limit the number of flow routes that can be in effect.

Use the `maximum-prefixes` statement to place a limit on the number of flow routes that can be installed in the `inetflow.0` RIB:

```
set routing-options rib inetflow.0 maximum-prefixes <number>
set routing-options rib inetflow.0 maximum-prefixes threshold <percent>
```

To limit the number of flow-spec routes permitted from a given BGP peer, use the `prefix-limit` statement for the flow family:

```
set protocols bgp group x neighbor <address> family inet flow prefix-limit
  maximum <number>
set protocols bgp group x neighbor <address> family inet flow prefix-limit
  teardown <%>
```

Summary

The Junos BGP flow-specification feature is a powerful tool against DDoS attacks that works well alongside your routing engine protection filters and the Trio DDoS prevention feature. Even a hardened control plane can be overrun with excessive traffic that is directed to a valid service such as SSH. Once alerted to the anomalous traffic, you can use flow-spec to rapidly deploy filters to all BGP speakers to eliminate the attack traffic as close to the source as possible, all the while being able to maintain connectivity to the router to perform such mitigation actions, thanks to your having the foresight to deploy best practice RE protection filters along with built-in DDoS protections via Trio FPCs.

BGP Flow-Specification Case Study

This section provides a sample use case for the BGP flow-spec feature. The network topology is shown in Figure 4-4.

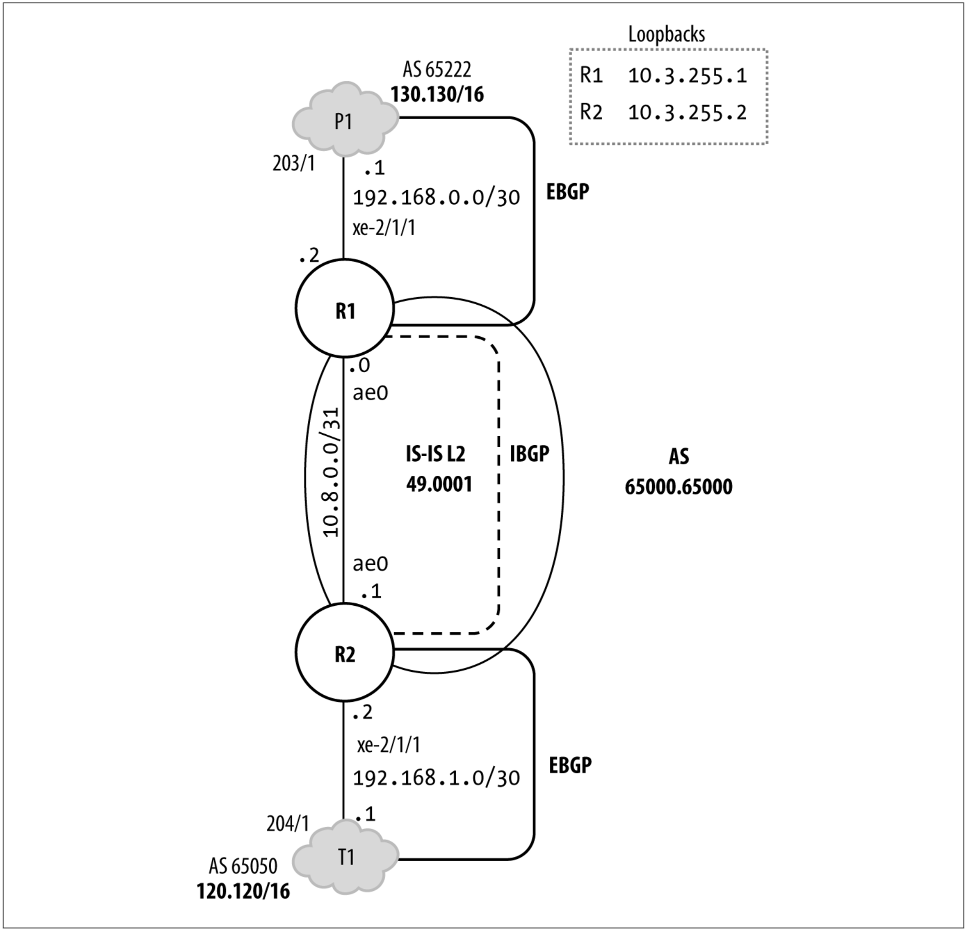


Figure 4-4. BGP Flow-Spec Topology.

Routers R1 and R2 have the best practice IPv4 RE protection filter previously discussed in effect on their loopback interfaces. The DDoS protection feature is enabled with the only change from the default being scaled FPC bandwidth for the ICMP aggregate. They peer with each other using loopback-based MP-IBGP, and to External peers P1 and T1 using EBGP. The P1 network is the source of routes from the 130.130/16 block, whereas T1 is the source of 120.120/16 routes. IS-IS Level 2 is used as the IGP. It runs passively on the external links to ensure the EBGP next-hops can be resolved. It is also used to distribute the loopback addresses used to support the IBGP peering. The protocols stanza on R1 is shown here:

```
{master}[edit]
jnpr@R1-RE0# show protocols
bgp {
    log-updown;
```

```
        group t1_v4 {
            type external;
            export bgp_export;
            peer-as 65050;
            neighbor 192.168.1.1;
        }
        group int_v4 {
            type internal;
            local-address 10.3.255.2;
            family inet {
                unicast;
                flow;
            }
            bfd-liveness-detection {
                minimum-interval 150;
                multiplier 3;
            }
            neighbor 10.3.255.1;
        }
    }
    isis {
        reference-bandwidth 100g;
        level 1 disable;
        interface xe-2/1/1.0 {
            passive;
        }
        interface ae0.1 {
            point-to-point;
            bfd-liveness-detection {
                minimum-interval 150;
                multiplier 3;
            }
        }
        interface lo0.0 {
            passive;
        }
    }
    lacp {
        traceoptions {
            file lacp_trace size 10m;
            flag process;
            flag startup;
        }
    }
    lldp {
        interface all;
    }
    layer2-control {
        nonstop-bridging;
    }
    vstp {
        interface xe-0/0/6;
        interface ae0;
        interface ae1;
        interface ae2;
        vlan 100 {
            bridge-priority 4k;
            interface xe-0/0/6;
            interface ae0;
            interface ae1;
            interface ae2;
        }
        vlan 200 {
            bridge-priority 8k;
            interface ae0;
            interface ae1;
            interface ae2;
        }
    }
```

Note that flow NLRI has been enabled for the inet family on the internal peering session. Note again that BGP flow-spec is also supported for EBGP peers, which means it can operate across AS boundaries when both networks have bilaterally agreed to support the functionality. The DDoS stanza is displayed here:

```
{master}[edit]
jnpr@R2-RE0# show system ddos-protection
protocols {
    icmp {
        aggregate {
            fpc 2 {
                bandwidth-scale 30;
                burst-scale 30;
            }
        }
    }
}
```

The DDoS settings alter FPC slot 2 to permit 30% of the system aggregate for ICMP. Recall from the previous DDoS section that by default all FPCs inherit 100% of the system aggregate, which means any one FPC can send at the full maximum load with no drops, but also means a DDoS attack on any one FPC can cause contention at aggregation points for other FPCs with normal loads. Here, FPC 2 is expected to permit some 6,000 PPS before it begins enforcing DDoS actions at 30% of the system aggregate, which by default is 20,000 PPS in this release.

You next verify the filter chain application to the lo0 interface. While only R1 is shown, R2 also has the best practice IPv4 RE protection filters in place; the operation of the RE protection filter was described previously in the RE protection case study.

```
{master}[edit]
jnpr@R1-RE0# show interfaces lo0
unit 0 {
    family inet {
        filter {
           input-list [ discard-frags accept-common-services accept-sh-bfd accept-bgp
accept-ldp accept-rsvp accept-telnet accept-vrrp discard-all ];
        }
        address 10.3.255.1/32;
    }
    family iso {
        address 49.0001.0100.0325.5001.00;
    }
    family inet6 {
        filter {
           input-list [ discard-extension-headers accept-MLD-hop-by-hop_v6 deny-icmp6-
undefined accept-common-services-v6 accept-sh-bfd-v6 accept-bgp-v6 accept-telnet-v6
accept-ospf3 accept-radius-v6 discard-all-v6 ];
        }
        address 2001:db8:1::ff:1/128;
    }
}
```

The IBGP and EBGP session status is confirmed. Though not shown, R2 also has both its neighbors in an established state:

```
{master}[edit]
jnpr@R1-RE0# run show bgp summary
Groups: 2 Peers: 2 Down peers: 0
Table          Tot Paths  Act Paths Suppressed    History Damp State    Pending
inet.0               200        200          0          0          0          0
inetflow.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...
10.3.255.2      65000.65000         12         12       0       0        3:14 Establ
  inet.0: 100/100/100/0
  inetflow.0: 0/0/0/0
192.168.0.1           65222          7         16       0       0        3:18 Establ
  inet.0: 100/100/100/
```

As is successful negotiation of the flow NLRI during BGP, capabilities exchange is confirmed by displaying the IBGP neighbor to confirm that the `inet-flow` NLRI is in effect:

```
{master}[edit]
jnpr@R1-RE0# run show bgp neighbor 10.3.255.2 | match nlri
  NLRI for restart configured on peer: inet-unicast inet-flow
  NLRI advertised by peer: inet-unicast inet-flow
  NLRI for this session: inet-unicast inet-flow
  NLRI that restart is negotiated for: inet-unicast inet-flow
  NLRI of received end-of-rib markers: inet-unicast inet-flow
  NLRI of all end-of-rib markers sent: inet-unicast inet-flow
```

And lastly, a quick confirmation of routing to both loopback and EBGP prefixes, from the perspective of R2:

```
{master}[edit]
jnpr@R2-RE0# run show route 10.3.255.1

inet.0: 219 destinations, 219 routes (219 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

10.3.255.1/32      *[IS-IS/18] 00:11:21, metric 5
                    > to 10.8.0.0 via ae0.1
```

```
{master}[edit]
jnpr@R2-RE0# run show route 130.130.1.0/24

inet.0: 219 destinations, 219 routes (219 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

130.130.1.0/24     *[BGP/170] 00:01:22, localpref 100, from 10.3.255.1
                      AS path: 65222 ?
                    > to 10.8.0.0 via ae0.1
```

The output confirms that IS-IS is supporting the IBGP session by providing a route to the remote router's loopback address, and that R2 is learning the 130.130/16 prefixes from R1, which in turn learned them via its EBGP peering to P1.

Let the Attack Begin!

With the stage set, things begin with a DDoS log alert at R2:

```
{master}[edit]
jnpr@R2-RE0# run show log messages | match ddos
Mar 18 17:43:47  R2-RE0 jddosd[75147]: DDOS_PROTOCOL_VIOLATION_SET: Protocol
ICMP:aggregate is violated at fpc 2 for 4 times, started at 2012-03-18 17:43:47 PDT,
last seen at 2012-03-18 17:43:47 PDT
```

Meanwhile, back at R1, no violations are reported, making it clear that R2 is the sole victim of the current DDoS bombardment:

```
{master}[edit]
jnpr@R1-RE0# run show ddos-protection protocols violations
Number of packet types that are being violated: 0
```

The syslog entry warns of excessive ICMP traffic at FPC 2. Details on the current violation are obtained with a `show ddos protocols` command:

```
jnpr@R2-RE0# run show ddos-protection protocols violations
Number of packet types that are being violated: 1
Protocol    Packet      Bandwidth  Arrival   Peak      Policer bandwidth
group       type        (pps)      rate(pps) rate(pps) violation detected at
icmp        aggregate   20000      13587     13610     2012-03-18 17:43:47 PDT
          Detected on: FPC-2

{master}[edit]
jnpr@R2-RE0# run show ddos-protection protocols icmp
Protocol Group: ICMP

  Packet type: aggregate (Aggregate for all ICMP traffic)
    Aggregate policer configuration:
      Bandwidth:        20000 pps
      Burst:            20000 packets
      Priority:         high
      Recover time:     300 seconds
      Enabled:          Yes
    System-wide information:
      Aggregate bandwidth is being violated!
```

```
        No. of FPCs currently receiving excess traffic: 1
        No. of FPCs that have received excess traffic:  1
        Violation first detected at: 2012-03-18 17:43:47 PDT
        Violation last seen at:      2012-03-18 17:58:26 PDT
        Duration of violation: 00:14:39 Number of violations: 4
      Received:  22079830            Arrival rate:     13607 pps
      Dropped:   566100              Max arrival rate: 13610 pps
    Routing Engine information:
      Aggregate policer is never violated
      Received:  10260083            Arrival rate:     6001 pps
      Dropped:   0                   Max arrival rate: 6683 pps
        Dropped by individual policers: 0
    FPC slot 2 information:
      Bandwidth: 30% (6000 pps), Burst: 30% (6000 packets), enabled
      Aggregate policer is currently being violated!
        Violation first detected at: 2012-03-18 17:43:47 PDT
        Violation last seen at:      2012-03-18 17:58:26 PDT
        Duration of violation: 00:14:39 Number of violations: 4
      Received:  22079830            Arrival rate:     13607 pps
      Dropped:   566100              Max arrival rate: 13610 pps
        Dropped by individual policers: 0
        Dropped by aggregate policer: 566100
```

The output confirms ICMP aggregate-level discards at the FPC level, with a peak load of 13,600 PPS, well in excess of the currently permitted 6,000 PPS. In addition, and much to your satisfaction, R2 remains responsive showing that the DDoS first line of defense is doing its job. However, aside from knowing there is a lot of ICMP arriving at FPC 2 for this router, there is not much to go on yet as far as tracking the attack back toward its source, flow-spec style or otherwise. You know this ICMP traffic must be destined for R2, either due to unicast or broadcast, because only host-bound traffic is subjected to DDoS policing.

The loopback filter counters and policer statistics are displayed at R2:

```
{master}[edit]
jnpr@R2-RE0# run show firewall filter lo0.0-i

Filter: lo0.0-i
Counters:
Name                                                Bytes              Packets
accept-bfd-lo0.0-i                                  25948                  499
accept-bgp-lo0.0-i                                   1744                   29
accept-dns-lo0.0-i                                      0                    0
accept-icmp-lo0.0-i                              42252794               918539
accept-ldp-discover-lo0.0-i                             0                    0
accept-ldp-igmp-lo0.0-i                                 0                    0
accept-ldp-unicast-lo0.0-i                              0                    0
accept-ntp-lo0.0-i                                      0                    0
accept-ntp-server-lo0.0-i                               0                    0
accept-rsvp-lo0.0-i                                     0                    0
accept-ssh-lo0.0-i                                      0                    0
accept-telnet-lo0.0-i                                7474                  180
accept-tldp-discover-lo0.0-i                            0                    0
accept-traceroute-icmp-lo0.0-i                          0                    0
```

```
    accept-traceroute-tcp-lo0.0-i                           0                    0
    accept-traceroute-udp-lo0.0-i                           0                    0
    accept-vrrp-lo0.0-i                                  3120                   78
    accept-web-lo0.0-i                                      0                    0
    discard-all-TTL_1-unknown-lo0.0-i                       0                    0
    discard-icmp-lo0.0-i                                    0                    0
    discard-ip-options-lo0.0-i                             32                    1
    discard-netbios-lo0.0-i                                 0                    0
    discard-tcp-lo0.0-i                                     0                    0
    discard-udp-lo0.0-i                                     0                    0
    discard-unknown-lo0.0-i                                 0                    0
    no-icmp-fragments-lo0.0-i                               0                    0
    Policers:
    Name                                                Bytes              Packets
    management-1m-accept-dns-lo0.0-i                        0                    0
    management-1m-accept-ntp-lo0.0-i                        0                    0
    management-1m-accept-ntp-server-lo0.0-i                 0                    0
    management-1m-accept-telnet-lo0.0-i                     0                    0
    management-1m-accept-traceroute-icmp-lo0.0-i            0                    0
    management-1m-accept-traceroute-tcp-lo0.0-i             0                    0
    management-1m-accept-traceroute-udp-lo0.0-i             0                    0
    management-5m-accept-icmp-lo0.0-i             21870200808            475439148
    management-5m-accept-ssh-lo0.0-i                        0                    0
    management-5m-accept-web-lo0.0-i                        0                    0
```

The counters for the `management-5m-accept-icmp-lo0.0-I` prefix-specific counter and policers make it clear that a large amount of ICMP traffic is hitting the loopback filter and being policed by the related 5 M policer. Given that the loopback policer is executed before the DDoS processing, right as host-bound traffic arrives at the Trio PFE, it's clear that the 5 Mbps of ICMP that is permitted by the policer amounts to more than the 6,000 PPS; otherwise, there would be no current DDoS alert or DDoS discard actions in the FPC.

Knowing that a policer evoked through a loopback filter is executed before any DDoS processing should help in dimensioning your DDoS and loopback policers so they work well together. Given that a filter-evoked policer measures bandwidth in bits per second while the DDoS policers function on a packet-per-second basis should make it clear that trying to match them is difficult at best and really isn't necessary anyway.

Because a loopback policer represents a system-level aggregate, there is some sense to setting the policer higher than that in any individual FPC. If the full expected aggregate arrives on a single FPC, then the lowered DDoS settings in the FPC will kick in to ensure that no one FPC can consume the system's aggregate bandwidth, thereby ensuring plenty of capacity of other FPCs that have normal traffic loads. The downside to such a setting is that you can now expect FPC drops even when only one FPC is active and below the aggregate system load.

Determine Attack Details and Define Flow Route

Obtaining the detail needed to describe the attack flow is where sampling or filter-based logging often come into play. In fact, the current RE protection filter has a provision for logging:

```
{master}[edit]
jnpr@R2-RE0# show firewall family inet filter accept-icmp
apply-flags omit;
term no-icmp-fragments {
    from {
        is-fragment;
        protocol icmp;
    }
    then {
        count no-icmp-fragments;
        log;
        discard;
    }
}
term accept-icmp {
    from {
        protocol icmp;
        ttl-except 1;
        icmp-type [ echo-reply echo-request time-exceeded unreachable source-quench
router-advertisement parameter-problem ];
    }
    then {
        policer management-5m;
        count accept-icmp;
        log;
        accept;
    }
}
```

The presence of the `log` and `syslog` action modifiers in the `accept-icmp` filter means you simply need to display the firewall cache or syslog to obtain the details needed to characterize the attack flow:

```
jnpr@R2-RE0# run show firewall log
Log :
Time      Filter    Action Interface     Protocol    Src Addr          Dest Addr
18:47:47  pfe       A      ae0.1         ICMP        130.130.33.1      10.3.255.2
18:47:47  pfe       A      ae0.1         ICMP        130.130.60.1      10.3.255.2
18:47:47  pfe       A      ae0.1         ICMP        130.130.48.1      10.3.255.2
18:47:47  pfe       A      ae0.1         ICMP        130.130.31.1      10.3.255.2
18:47:47  pfe       A      ae0.1         ICMP        130.130.57.1      10.3.255.2
18:47:47  pfe       A      ae0.1         ICMP        130.130.51.1      10.3.255.2
18:47:47  pfe       A      ae0.1         ICMP        130.130.50.1      10.3.255.2
18:47:47  pfe       A      ae0.1         ICMP        130.130.3.1       10.3.255.2
18:47:47  pfe       A      ae0.1         ICMP        130.130.88.1      10.3.255.2
18:47:47  pfe       A      ae0.1         ICMP        130.130.94.1      10.3.255.2
18:47:47  pfe       A      ae0.1         ICMP        130.130.22.1      10.3.255.2
18:47:47  pfe       A      ae0.1         ICMP        130.130.13.1      10.3.255.2
18:47:47  pfe       A      ae0.1         ICMP        130.130.74.1      10.3.255.2
```

```
18:47:47  pfe       A      ae0.1         ICMP        130.130.77.1        10.3.255.2
18:47:47  pfe       A      ae0.1         ICMP        130.130.46.1        10.3.255.2
18:47:47  pfe       A      ae0.1         ICMP        130.130.94.1        10.3.255.2
18:47:47  pfe       A      ae0.1         ICMP        130.130.38.1        10.3.255.2
18:47:47  pfe       A      ae0.1         ICMP        130.130.36.1        10.3.255.2
18:47:47  pfe       A      ae0.1         ICMP        130.130.47.1        10.3.255.2
. . .
```

The contents of the firewall log make it clear the attack is ICMP based (as already known), but in addition you can now confirm the destination address matches R2's loopback, and that the source appears to be from a range of 130.130.x/24 subnets from within P1's 130.130/16 block. Armed with this information, you can contact the administrator of the P1 network to ask them to address the issue, but that can wait until you have this traffic filtered *at ingress* to your network, rather than *after* it has had the chance to consume resources in your network and at R2, specifically.

A flow route is defined on R2:

```
{master}[edit]
jnpr@R2-RE0# show routing-options flow
route block_icmp_p1 {
    match {
        destination 10.3.255.2/32;
        source 130.130.0.0/16;
        protocol icmp;
    }
    then discard;
}
```

The flow matches all ICMP traffic sent to R2's loopback address from any source in the 130.130/16 space with a discard action. Once locally defined, the flow-spec is placed into effect (there is no validation for a local flow-spec, much like there is no need to validate a locally defined firewall filter), as confirmed by the current DDoS statistics, which now report a 0 PPs arrival rate:

```
{master}[edit]
jnpr@R2-RE0# run show ddos-protection protocols icmp
Protocol Group: ICMP

  Packet type: aggregate (Aggregate for all ICMP traffic)
    Aggregate policer configuration:
      Bandwidth:        20000 pps
      Burst:            20000 packets
      Priority:         high
      Recover time:     300 seconds
      Enabled:          Yes
    System-wide information:
      Aggregate bandwidth is no longer being violated
        No. of FPCs that have received excess traffic: 1
        Last violation started at: 2012-03-18 18:47:28 PDT
        Last violation ended at:   2012-03-18 18:52:59 PDT
        Duration of last violation: 00:05:31 Number of violations: 5
      Received:  58236794            Arrival rate:     0 pps
      Dropped:   2300036             Max arrival rate: 13620 pps
```

```
Routing Engine information:
  Aggregate policer is never violated
  Received:  26237723           Arrival rate:     0 pps
  Dropped:   0                  Max arrival rate: 6683 pps
    Dropped by individual policers: 0
FPC slot 2 information:
  Bandwidth: 30% (6000 pps), Burst: 30% (6000 packets), enabled
  Aggregate policer is no longer being violated
    Last violation started at: 2012-03-18 18:47:28 PDT
    Last violation ended at:   2012-03-18 18:52:59 PDT
    Duration of last violation: 00:05:31 Number of violations: 5
  Received:  58236794           Arrival rate:     0 pps
  Dropped:   2300036            Max arrival rate: 13620 pps
    Dropped by individual policers: 0
    Dropped by aggregate policer: 2300036
```

The presence of a flow-spec filter is confirmed with a `show firewall` command:

```
{master}[edit]
jnpr@R2-RE0# run show firewall | find flow

Filter: __flowspec_default_inet__
Counters:
Name                                                Bytes              Packets
10.3.255.2,130.130/16,proto=1                127072020948           2762435238
```

The presence of the flow-spec filter is good, but the non-zero counters confirm that it's still matching a boatload of traffic to 10.3.255.2, from 130.130/16 sources, for protocol 1 (ICMP), as per its definition. Odd, as in theory R1 should now also be filtering this traffic, which clearly is not the case; more on that later.

It's also possible to display the `inetflow.0` table directly to see both local and remote entries; the table on R2 currently has only its one locally defined flow-spec:

```
{master}[edit]
jnpr@R2-RE0# run show route table inetflow.0 detail

inetflow.0: 1 destinations, 1 routes (1 active, 0 holddown, 0 hidden)
10.3.255.2,130.130/16,proto=1/term:1 (1 entry, 1 announced)
        *Flow   Preference: 5
                Next hop type: Fictitious
                Address: 0x8df4664
                Next-hop reference count: 1
                State: <Active>
                Local AS: 4259905000
                Age: 8:34
                Task: RT Flow
                Announcement bits (2): 0-Flow 1-BGP_RT_Background
                AS path: I
                Communities: traffic-rate:0:0
```

Don't be alarmed about the fictitious next-hop bit. It's an artifact from the use of BGP, which has a propensity for next-hops, versus a flow-spec, which has no such need. Note also how the `discard` action is conveyed via a community that encodes an action of rate limiting the matching traffic to 0 bps.

With R2 looking good, let's move on to determine why R1 is apparently not yet filtering this flow. Things begin with a confirmation that the flow route is advertised to R1:

```
{master}[edit]
jnpr@R2-RE0# run show route advertising-protocol bgp 10.3.255.1 table inetflow.0

inetflow.0: 1 destinations, 1 routes (1 active, 0 holddown, 0 hidden)
  Prefix                  Nexthop              MED     Lclpref    AS path
  10.3.255.2,130.130/16,proto=1/term:1
*                         Self                         100        I
```

As expected, the output confirms that R2 is sending the flow-spec to R1, so you expect to find a matching entry in its `inetflow.0` table, along with a dynamically created filter that should be discarding the attack traffic at ingress from P1 as it arrives on the xe-2/1/1 interface. But, thinking back, it was noted that R2's local flow route is showing a high packet count and discard rate, which clearly indicates that R1 is still letting this traffic through.

Your curiosity piqued, you move to R1 and find the flow route is hidden:

```
{master}[edit]
jnpr@R1-RE0# run show route table inetflow.0 hidden detail

inetflow.0: 1 destinations, 1 routes (0 active, 0 holddown, 1 hidden)
10.3.255.2,130.130/16,proto=1/term:N/A (1 entry, 0 announced)
         BGP                /-101
                Next hop type: Fictitious
                Address: 0x8df4664
                Next-hop reference count: 1
                State: <Hidden Int Ext>
                Local AS: 65000.65000 Peer AS: 65000.65000
                Age: 16:19
                Task: BGP_65000.65000.10.3.255.2+179
                AS path: I
                Communities: traffic-rate:0:0
                Accepted
                Validation state: Reject, Originator: 10.3.255.2
                Via: 10.3.255.2/32, Active
                Localpref: 100
                Router ID: 10.3.255.2
```

Given the flow route is hidden, no filter has been created at R1:

```
{master}[edit]
jnpr@R1-RE0# run show firewall | find flow

Pattern not found
{master}[edit]
```

And as a result, the attack data is confirmed to be leaving R1's ae0 interface on its way to R2:

```
Interface: ae0, Enabled, Link is Up
Encapsulation: Ethernet, Speed: 20000mbps
Traffic statistics:                                          Current delta
  Input bytes:                 158387545 (6624 bps)                    [0]
  Output bytes:            4967292549831 (2519104392 bps)                [0]
  Input packets:                 2335568 (12 pps)                        [0]
  Output packets:            52813462522 (6845389 pps)                   [0]
Error statistics:
  Input errors:                        0                                 [0]
  Input drops:                         0                                 [0]
  Input framing errors:                0                                 [0]
  Carrier transitions:                 0                                 [0]
  Output errors:                       0                                 [0]
  Output drops:                        0                                 [0]
```

Thinking a bit about the hidden flow-spec and its rejected state, the answer arrives: this is a validation failure. Recall that, by default, only the current best source of a route is allowed to generate a flow-spec that could serve to filter the related traffic. Here, R2 is not the BGP source of the 130.130/16 route that the related flow-spec seeks to filter. In effect, this is a third-party flow-spec, and as such, it does not pass the default validation procedure. You can work around this issue by using the `no-validate` option along with a policy at R1 that tells it to accept the route. First, the policy:

```
{master}[edit]
jnpr@R1-RE0# show policy-options policy-statement accept_icmp_flow_route
term 1 {
    from {
        route-filter 10.3.255.2/32 exact;
    }
    then accept;
}
```

The policy is applied under the flow family using the `no-validate` keyword:

```
{master}[edit]
jnpr@R1-RE0# show protocols bgp group int
type internal;
local-address 10.3.255.1;
family inet {
    unicast;
    flow {
        no-validate accept_icmp_flow_route;
    }
}
bfd-liveness-detection {
    minimum-interval 2500;
    multiplier 3;
}
neighbor 10.3.255.2
```

After the change is committed, the flow route is confirmed at R1:

```
{master}[edit]
jnpr@R1-RE0# run show firewall | find flow

Filter: __flowspec_default_inet__
Counters:
Name                                                Bytes              Packets
10.3.255.2,130.130/16,proto=1                  9066309970            197093695
```

The 10.3.255.2,130.130/16,proto=1 flow-spec filter has been activated at R1, a good indication the flow-spec route is no longer hidden due to validation failure. The net result that you have worked so hard for is that now, the attack data is no longer being transported over your network just to be discarded at R2.

```
{master}[edit]
jnpr@R1-RE0# run monitor interface ae0

Next='n', Quit='q' or ESC, Freeze='f', Thaw='t', Clear='c', Interface='i'
R1-RE0                            Seconds: 0                     Time: 19:18:22
                                                          Delay: 16/16/16
Interface: ae0, Enabled, Link is Up
Encapsulation: Ethernet, Speed: 20000mbps
Traffic statistics:                                           Current delta
  Input bytes:                  158643821 (6480 bps)                    [0]
  Output bytes:             5052133735948 (5496 bps)                    [0]
  Input packets:                  2339427 (12 pps)                      [0]
  Output packets:             54657835299 (10 pps)                      [0]
Error statistics:
  Input errors:                         0                               [0]
  Input drops:                          0                               [0]
  Input framing errors:                 0                               [0]
  Carrier transitions:                  0                               [0]
  Output errors:                        0                               [0]
  Output drops:                         0                               [0]
```

This completes the DDoS mitigation case study.

Summary

The Junos OS combined with Trio-based PFEs offers a rich set of stateless firewall filtering, a rich set of policing options, and some really cool built-in DDoS capabilities. All are performed in hardware so you can enable them in a scaled production environment without appreciable impact to forwarding rates.

Even if you deploy your MX in the core, where edge-related traffic conditions and contract enforcement is typically not required, you still need stateless filters, policers, and/or DDoS protection to protect your router's control plane from unsupported services and to guard against excessive traffic, whether good or bad, to ensure the router remains secure and continues to operate as intended even during periods of abnormal volume of control plane traffic, be it intentional or attack based.

This chapter provided current best practice templates from strong RE protection filters for both IPv4 and IPv6 control plane. All readers should compare their current RE protection filters to the examples provided to decide if any modifications are needed to maintain current best practice in this complex, but all too important, subject. The new DDoS feature, supported on Trio line cards only, works symbiotically with RE protection filters, or can function standalone, and acts as a robust primary, secondary, and tertiary line of defense to protect the control plane from resource exhaustion, stemming from excessive traffic that could otherwise impact service, or worse, render the device inoperable and effectively unreachable during the very times you need access the most!

Chapter Review Questions

1. Which is true regarding the DDoS prevention feature?
   a. The feature is off by default
   b. The feature is on by default with aggressive policers
   c. The feature is on by default but requires policer configuration before any alerts or policing can occur
   d. The feature is on by default with high policer rates that in most cases exceed system control plane capacity to ensure no disruption to existing functionality
2. Which is true about DDoS policers and RE protection policers evoked though a filter?
   a. The lo0 policer is disabled when DDoS is in effect
   b. The DDoS policers run first with the lo0 policer executed last
   c. The lo0 policer is executed before and after the DDoS policers, once at ingress and again in the RE
   d. Combining lo0 and DDoS policers is not permitted and a commit error is retuned
3. A strong RE protection filter should end with which of the following?
   a. An accept all to ensure no disruption
   b. A reject all, to send error messages to sources of traffic that is not permitted
   c. A discard all to silently discard traffic that is not permitted
   d. A log action to help debug filtering of valid/permitted services
   e. Both C and D
4. A filter is applied to the main instance lo0.0 and a VRF is defined without its own lo0.n ifl. Which is true?
   a. Traffic from the instance to the local control plane is filtered by the lo0.0 filter
   b. Traffic from the instance to remote VRF destinations is filtered by the lo0.0 filter
   c. Traffic from the instance to the local control plane is not filtered
   d. None of the above. VRFs require a lo0.n for their routing protocols to operate
5. What Junos feature facilitates simplified filter management when using address-based match criteria to permit only explicitly defined BGP peers?
   a. Dynamic filter lists
   b. Prefix lists and the `apply-path` statement

   c. The ability to specify a 0/0 as a match-all in an address-based match condition
   d. All of the above
   e. A sr-TCM policer applied at the unit level for all Layer 2 families using the `layer2-policer` statement

6. What is the typical use case for an RE filter applied in the output direction?
   a. To ensure your router is not generating attack traffic
   b. To track the traffic sent from the router for billing purposes
   c. A trick question; output filters are not supported
   d. To alter CoS/ToS marking and queuing for locally generated control plane traffic

Chapter Review Answers

1. **Answer: D**. Because DDoS is on by default, the policers are set to the same high values as when the feature is disabled, effectively meaning the host-bound traffic from a single PFE is limited by the processing path capabilities and not DDoS protection. You must reduce the defaults to suit the needs of your network to gain additional DDoS protection outside of alerting and policing at aggregation points for attacks on multiple PFEs.
2. **Answer: C**. When an lo0 policer is present, it is executed first, as traffic arrives at the line card, before any DDoS (even Trio PFE-level) are executed. In addition, a copy of the RE policer is also stored in the kernel where its acts on the aggregate load going to the RE, after the DDoS policer stage.
3. **Answer: E**. A strong security filter always uses a discard-all as a final term. Using rejects can lead to resource usage in the form of error messages, a bad idea when under an attack. Adding the log action to the final term is a good idea, as it allows you to quickly confirm what traffic is hitting the final discard term. Unless you are being attacked, very little traffic should be hitting the final term, so the log action does not represent much burden. The firewall cache is kept in kernel, and only displayed when the operator requests the information, unlike a syslog filter action, which involves PFE-to-RE traffic on an ongoing basis for traffic matching the final discard term.
4. **Answer: A**. When a routing instance has filter applied to an lo0 unit in that instance, that filter is used; otherwise, control plane traffic from the instance to the RE is filtered by the main instance lo0.0 filter.
5. **Answer: B**. Use `prefix-lists` and the `apply-path` feature to build a dynamic list of prefixes that are defined somewhere else on the router (e.g., those assigned to interfaces or used in BGP peer definitions), and then use the dynamic list as a match condition in a filter to simplify filter management in the face of new interface or peer definitions.

6. **Answer: D**. Output filters are most often used to alter the default CoS/ToS marking for locally generated traffic.

CHAPTER 5

Trio Class of Service

This chapter explores the vast and wonderful world of Class of Service (CoS) on Trio-based PFEs. A significant portion of the chapter is devoted to its highly scalable hierarchical CoS (H-CoS) capabilities.

Readers that are new to general IP CoS processing in Junos or who desire a review of IP Differentiated Services (DiffServ) concepts should consult the *Juniper Enterprise Routing* book. CoS is a complicated subject, and the intent is to cover new, Trio-specific CoS capabilities without dedicating invaluable space to material that is available elsewhere.

The CoS topics in this chapter include:

- MX router CoS capabilities
- Trio CoS flow
- Hierarchical CoS
- Trio scheduling, priority handling, and load balancing
- MX CoS defaults
- Predicting queue throughput
- Per-unit Scheduling CoS lab
- Hierarchical CoS lab

MX CoS Capabilities

This section provides an overview of Trio-based MX router CoS capabilities and operation, which includes a packet walkthrough that illustrates how CoS is provided to transit traffic in the Trio architecture.

The Trio chipset offers several CoS differentiators that should be kept in mind when designing your network's CoS architecture.

Intelligent Class Aware Hierarchical Rate Limiters

This feature lets Trio PFEs honor user configured rate limit (shaping) policies for multiple classes of traffic, while at the same time protecting conforming high-priority traffic from low-priority traffic bursts. This is accomplished though support of up to four levels of scheduling and queuing (Ports/IFL sets/IFLs/Queues), with support for a shaping rate (PIR), a guaranteed rate (CIR), and excess rate control at all levels.

Additionally, you have great flexibility as to the attachment points for hierarchical scheduling and shaping, to include IFDs (ports), IFLs (logical interfaces), and interface sets (which are collections of IFLs or VLANs, and the key enabler of H-CoS).

Priority-Based Shaping

This feature allows you to shape traffic at an aggregate level based on its priority level, either at the port or IFL-set levels. Priority-based shaping is well suited to broadband aggregation where large numbers of individual flows are combined into larger class-based aggregates, which can now be shaped at a macro level.

Dynamic Priority Protection

Priority inheritance combined with the ability to demote or promote the priority of a traffic class protects bandwidth of high-priority traffic even in the presence of bursty low-priority traffic.

Highly Scalable CoS

Trio PFEs offer scalable CoS that ensures your hardware investment can grow to meet current and future CoS need. Key statistics include up to 512k queues per MPC slot, up to 64k VLANs with eight queues attached per MPC slot, and up to 16k VLAN groups per MPC slot.

Dynamic CoS Profiles

Dynamic CoS allows MX platforms to provide a customized CoS profile for PPPOE/DHCP/L2TP, etc. Subscriber access where RADIUS authentication extension can include CoS parameters that, for example, might add an EF queue to a triple play subscriber for the duration of some special event.

Port versus Hierarchical Queuing MPCs

In general, MX routers carry forward preexisting Junos CoS capabilities while adding numerous unique capabilities. Readers looking for a basic background in Junos CoS capability and configuration are encouraged to consult the *Juniper Enterprise Routing* book. From a CoS perspective, Trio-based MX platforms support two categories of line cards, namely, those that do only port-level CoS and those that can provide hierarchical CoS (H-CoS). The latter types provide fine-grained queuing and additional levels of scheduling hierarchy, as detailed later, and are intended to meet the needs of broadband subscriber access where CoS handling is needed for literally thousands of users.

Port-based queuing MPCs support eight queues (per port) and also provide port-level shaping, per-VLAN (IFL) classification, rewrites, and policing.

Port-based MPC types include:

MX-MPC3E-3D
MPC-3D-16XGE-SFPP
MPC-3D-16XGE-SFPP-R-B
MX-MPC1-3D
MX-MPC1-3D-R-B
MX-MPC2-3D
MX-MPC2-3D-R-B

Hierarchical queuing MPCs support all port-level CoS functionality in addition to H-CoS, which adds a fourth level of hierarchy via the addition of an IFL set construct. Only H-CoS capable MPCs have the dense queuing block, which is currently facilitated by the QXASIC. H-CoS-capable MPCs include:

MX-MPC1-Q-3D
MX-MPC1-3D-Q-R-B
MX-MPC2-Q-3D
MX-MPC2-3D-Q-R-B
MX-MPC2-EQ-3D
MX-MPC2-3D-EQ-R-B

> This chapter demonstrates use of shell commands to illustrate operation and debugging steps. These commands are not officially supported and should only be used under guidance of JTAC. Incorrect usage of these commands can be service impacting.

A `show chassis hardware` command can be used to confirm of a MPC supports H-CoS; such MPCs are designated with a “Q”:

```
{master}[edit]
jnpr@R1-RE0# run show chassis hardware
Hardware inventory:
Item             Version  Part number  Serial number     Description
Chassis                                JN111992BAFC      MX240
Midplane         REV 07   760-021404   TR5026            MX240 Backplane
FPM Board        REV 03   760-021392   KE2411            Front Panel Display
PEM 0            Rev 02   740-017343   QCS0748A002       DC Power Entry Module
Routing Engine 0 REV 07   740-013063   1000745244        RE-S-2000
Routing Engine 1 REV 07   740-013063   9009005669        RE-S-2000
CB 0             REV 03   710-021523   KH6172            MX SCB
CB 1             REV 10   710-021523   ABBM2781          MX SCB
FPC 2            REV 15   750-031088   YR7184            MPC Type 2 3D Q
. . .
```

If there is ever any doubt, or if you are not sure how many QX chips your Trio MPC supports, you can access the MPC via VTY and display the dense queuing block’s driver. If driver details are returned, the card should be H-CoS capable. In the v11.4R1 release, the `show qxchip driver <n>` command is used, where n refers to the buffer manager number, which currently is either 0 or 1 as some MPC types support two buffer management blocks:

```
{master}[edit]
jnpr@R1-RE0# run start shell pfe network fpc2


NPC platform (1067Mhz MPC 8548 processor, 2048MB memory, 512KB flash)

NPC2(R1-RE0 vty)# NPC2(R1-RE0 vty)# show qxchip 0 driver
QX-chip : 0
   Debug flags           : 0x0
   hw initialized        : TRUE
   hw present            : TRUE
   q-drain-workaround    : Disabled
   periodics enabled     : TRUE

   rldram_size    : 603979776
   qdr_size       : 37748736

                    Scheduler 0                     Scheduler 1
                 Allocated   Maximum             Allocated   Maximum
                 -------------------             -------------------
L1                      3        63                     3        63
L2                      5      4095                     3      4095
L3                      8      8191                     3      8191
Q                      64     65532                    24     65532

Q Forced drain workaround Counter :      0
Q Forced drain workaround time:      0 us
Q BP drain workaround Counter :      4
Q stats msb notification count:      0
ISSU HW sync times:      0 ms
   sched block:      0 ms
   drop block:      0 ms
Drain-L1 node:      0 (sched0)      64 (sched1)
Drain-L2 node:      0 (sched0)    4096 (sched1)
Drain-L3 node:      0 (sched0)   16384 (sched1)
Drain-base-Q :      0 (sched0)  131072 (sched1)
```

To provide contrast, this output is from a nonqueuing MPC:

```
--- JUNOS 12.1R1.9 built 2012-03-24 12:52:33 UTC
jnpr@R3>show chassis hardware
Hardware inventory:
. . .
FPC 2            REV 14   750-031089   YF1316            MPC Type 2 3D
. . .

NPC2(R3 vty)# show qxchip 0 driver
QXCHIP 0 does not exist
```

H-CoS and the MX80

The MX5, 10, 40, and 80 platforms each contain a built-in routing engine and one Packet Forwarding Engine (PFE). The PFE has two "pseudo" Flexible PIC Concentrators (FPC 0 and FPC1). H-CoS is supported on these platforms, but currently only for the modular MIC slots labeled MIC0 and MIC1, which are both housed in FPC1. H-CoS is not supported on the four fixed 10xGE ports (which are usable on the MX40 and MX80 platforms), which are housed in FPC0.

H-CoS is not supported on the MX80-48T fixed chassis.

CoS versus QoS?

Many sources use the terms CoS and QoS interchangeably. To try and bring order to the cosmos, here CoS is used for the net effect, whereas QoS is reserved for describing individual parameters, such as end-to-end delay variation (jitter). The cumulative effects of the QoS parameters assigned to a user combine to form the *class of service* definition. Taking air travel as an example, first class is a class of service and can be characterized by a set of QoS parameters that include a big comfy seat, metal tableware, real food, etc. Different air carriers can assign different values to these parameters (e.g., better wines, more seat incline, etc.) to help differentiate their service levels from other airlines that also offer a first class service to try and gain a competitive advantage.

CoS Capabilities and Scale

Table 5-1 highlights key CoS capabilities for various MPC types.

Table 5-1. MPC CoS Feature Comparison.

Feature	MPC1, MPC2, 16x10G MPC	MPC1-Q	MPC2-Q	MPC2-EQ
Queuing	Eight queues per port	Eight queues per port and eight queues per VLAN	Eight queues per port and eight queues per VLAN	Eight queues per port and eight queues per VLAN
Port Shaping	Yes	Yes	Yes	Yes
Egress Queues	8 queues per port	128 k (64 k Ingress/64 k Egress)	256 k (128 k Ingress/ 128 k Egress)	512 K
Interface-sets (L2 scheduling nodes)	NA	8 K	8 K	16 K
Queue Shaping, Guaranteed Rate, and LLQ	CIR/PIR/LLQ	CIR/PIR/LLQ	CIR/PIR/LLQ	CIR/PIR/LLQ
VLAN shaping (per-unit scheduler)	NA	CIR/PIR	CIR/PIR	CIR/PIR

Feature	MPC1, MPC2, 16x10G MPC	MPC1-Q	MPC2-Q	MPC2-EQ
Interface Set Level Shaping	NA	CIR/PIR	CIR/PIR	CIR/PIR
WRED	Four profiles, uses Tail RED	Four profiles, uses Tail RED	Four profiles, uses Tail RED	Four profiles, uses Tail RED
Rewrite	MPLS EXP, IP Prec/ DSCP (egress or ingress), 802.1p inner/ outer	MPLS EXP, IP Prec/DSCP (egress or ingress), 802.1p inner/outer	MPLS EXP, IP Prec/DSCP (egress or ingress), 802.1p inner/outer	MPLS EXP, IP Prec/ DSCP (egress or ingress), 802.1p inner/ outer
Classifier (per VLAN/IFL)	MPLS EXP, IP Prec/ DSCP, 802.1p (inner and outer tag), Multi-Field	MPLS EXP, IP Prec/ DSCP, 802.1p (inner and outer tag), Multi-Field	MPLS EXP, IP Prec/ DSCP, 802.1p (inner and outer tag), Multi-Field	MPLS EXP, IP Prec/ DSCP, 802.1p (inner and outer tag), Multi-Field
Policer per VLAN/ IFL	Single rate two-color, srTCM, trTCM, hierarchical	Single rate two-color, srTCM, trTCM, hierarchical	Single rate two-color, srTCM, trTCM, hierarchical	Single rate two-color, srTCM, trTCM, hierarchical

Queue and Scheduler Scaling

Table 5-2 lists supported queue and subscriber limits for Trio MPCs. Note that the supported IFL numbers are per PIC. On PICs with multiple ports, the IFL counts should be dived among all ports for optimal scaling.

Table 5-2. MPC Queue and Subscriber Scaling.

MPC Type	Dedicated Queues	Subscribers/ IFLs	IFLs: Four Queues	IFLs: Eight Queues
30-Gigabit Ethernet Queuing MPC (MPC1-3D-Q)	64 k	16 Kk	16 k (8 k per PIC)	8 k (4 k per PIC)
60-Gigabit Ethernet Queuing MPC (MPC2-3D-Q)	128 k	32 k	32 k (8 k per PIC)	16 k (4 k per PIC)
60-Gigabit Ethernet Enhanced Queuing MPC (MPC2-3D-EQ)	512 k	64 k	64 k (16 k per PIC)	64 k (16 k per PIC)

Table 5-3 summarizes the currently supported scale for H-CoS on fine-grained queuing MPCs as of the v11.4R1 Junos release.

Capabilities constantly evolve, so always check the release notes and documentation for your hardware and Junos release to ensure you have the latest performance capabilities.

Table 5-3. Queue and Scheduler Node Scaling.

Feature	MPC1-Q	MPC2-Q	MPC2-EQ
Queues (Level 4)	128 k (split between ingress/ egress	256 k (split between ingress/ egress)	512 k
IFLs (Level 3)	Four queues: 16 k/8 k per PIC Eight queues: 8 K/4 k per PIC	Four queues: 32 k/8 k per PIC Eight queues: 16 K/4 k per PIC	Four or eight queues: 64 k/16 k per PIC
IFL-Set nodes (Level 2)	8 k	16 k	16 k
Port nodes (Level 1)	128	256	256

Though currently only egress queuing is supported, future Junos releases may support ingress queuing in an evolution path similar to the previous IQ2E cards, which also provided H-CoS. Note how the Q-type MPCs divide the pool of queues with half dedicated to ingress and egress pools, respectively. In contrast, the EQ MPC can use all 512 k queues for egress, or it can split the pool for ingress and egress use. Note again that in the v11.4R1 release, ingress queuing is not supported for Trio MPCs.

How Many Queues per Port? Knowing how many queues are supported per MPC and MIC is one thing. But, given that many MICs support more than one port, the next question becomes, "How many queues do I get per port?" The answer is a function of the number of Trio PFEs that are present on a given MPC.

For example, 30-gigabit Ethernet MPC modules have one PFE, whereas the 60-gigabit Ethernet MPC modules have two. Each PFE in turn has two scheduler blocks that share the management of the queues. On the MPC1-3D-Q line cards, each scheduler block maps to one-half of a MIC; in CLI configuration statements, that one-half of a MIC corresponds to one of the four possible PICs, numbered 0 to 3. MIC ports are partitioned equally across the PICs. A two-port MIC has one port per PIC. A four-port MIC has two ports per PIC.

Figure 5-1 shows the queue distribution on a 30-gigabit Ethernet queuing MPC module when both MPCs are populated to support all four PICs.

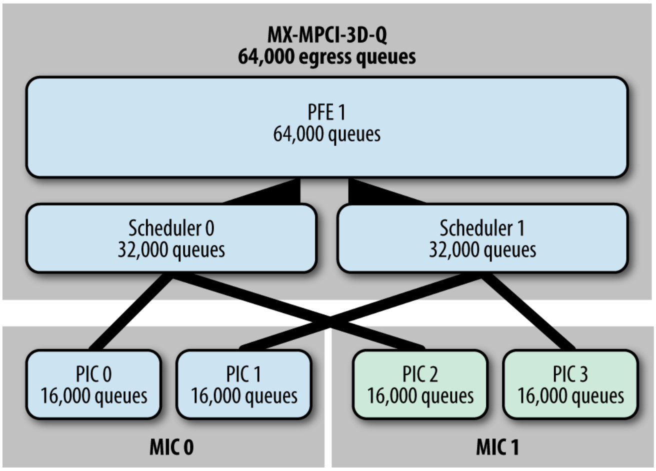


Figure 5-1. Queue Distribution on MPC1-3D-Q: Four PICs.

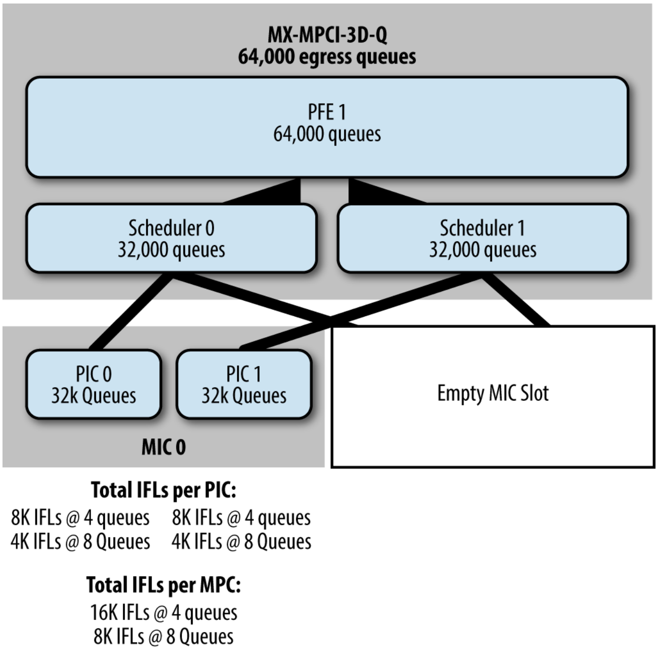


Figure 5-2. Queue Distribution on MPC1-3D-Q: Two PICs.

When all four PICs are installed, each scheduler maps to two PICs, each of which is housed on a different MIC. For example, scheduler 0 maps to PIC 0 on MIC 0 and to

PIC 2 on MIC 1, while scheduler 1 maps to PIC 1 on MIC 0 and to PIC 3 on MPC 1. One-half of the 64,000 egress queues are managed by each scheduler.

In this arrangement, one-half of the scheduler's total queue complement (16 k) is available to a given PIC. If you allocate four queues per IFL (subscriber), this arrangement yields 4 k IFLs per PIC; if desired, all PIC queues can be allocated to a single PIC port or spread over IFLs assigned to multiple PIC ports, but you cannot exceed 16 k queues per PIC. A maximum of 2 k IFLs can be supported per PIC when using eight queues per IFL.

Figure 5-2 shows another possible PIC arrangement for this MPC; in this case, one MIC is left empty to double the number of queues available on the remaining MIC.

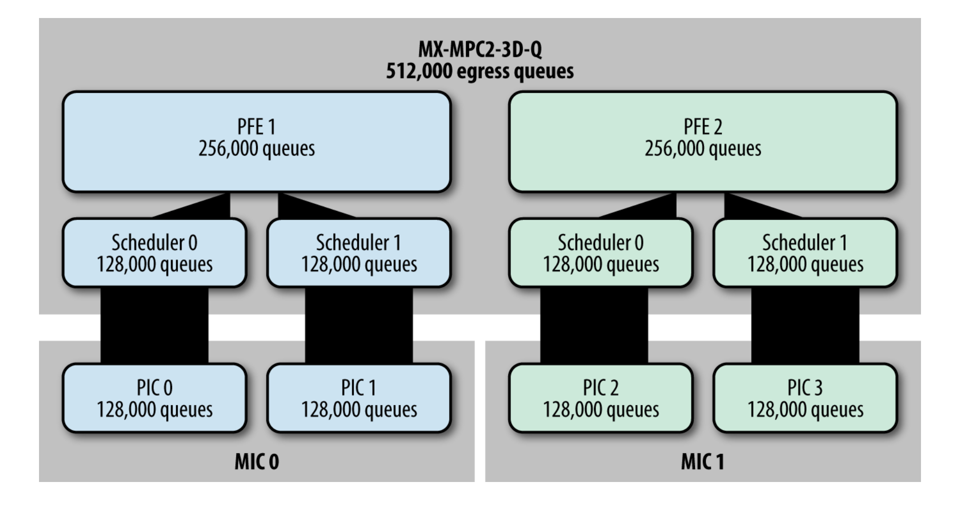


Figure 5-3. Queue Distribution on the 60-Gigabit Ethernet Queuing MPC Module.

By leaving one MIC slot empty, all 32 k queues are made available to the single PIC that is attached to each scheduler block. This arrangement does not alter the total MPC scale, which is still 16 k IFLs using four queues per subscriber; however, now you divide the pool of queues among half as many PICs/ports, which yields twice the number of subscribers per port, bringing the total to 8 k IFLs per PIC when in four-queue mode and 4 k in eight-queue mode.

On 60-gigabit Ethernet queuing and enhanced queuing Ethernet MPC modules, each scheduler maps to only one-half of a single MIC: PIC 0 or PIC 1 for the MIC in slot 0 and PIC 2 or PIC 3 for the MIC in slot 1. Figure 5-3 shows how queues are distributed on a 60-gigabit ethernet enhanced queuing MPC module.

Of the 512,000 egress queues possible on the module, one-half (256,000) are available for each of the two Packet Forwarding Engines. On each PFE, half of these queues (128,000) are managed by each scheduler. The complete scheduler complement

(128,000) is available to each PIC in the MIC. If you allocate all the queues from a scheduler to a single port, then the maximum number of queues per port is 128,000. If you dedicate four queues per subscriber, you can accommodate a maximum of 32,000 subscribers on a single MPC port. As before, half that if you provision eight queues per subscribers, bringing the maximum to 16,000 subscribers per MPC port.

The number of MICs installed and the number of ports per MIC does not affect the maximum number of queues available on a given port for this MPC type. This module supports a maximum of 64,000 subscribers regardless of whether you allocate four or eight queues per PIC. The MPC supports a maximum of 128,000 queues per port. If you have two two-port MICs installed, each PIC has one port and you can have 128,000 queues on each port. You can have fewer, of course, but you cannot allocate more to any port. If you have two four-port MICs installed, you can have 128,000 queues in each PIC, but only on one port in each PIC. Or you can split the queues available for the PIC across the two ports in each PIC.

Configure Four- or Eight-Queue Mode. Given that all Trio MPCs support eight queues, you may ask yourself, "how do I control how many queues are allocated to a given IFL?" Simply defining four or fewer forwarding classes (FCs) is not enough to avoid allocating eight queues, albeit with only four in use. When a four FC configuration is in effect, the output of a `show interfaces queue` command displays `Queues supported: 8, Queues in use: 4`, but it's important to note that the scheduler node still allocates eight queues from the available pool. To force allocation of only four queues per IFL, you must configure the maximum queues for that PIC as four at the `[edit chassis]` hierarchy:

```
jnpr@R1-RE0# show chassis
redundancy {
    graceful-switchover;
}
. . .
fpc 2 {
    pic 0 {
        max-queues-per-interface 4;
```

Changing the number of queues results in an MPC reset.

Low Queue Warnings. An SNMP trap is generated to notify you when the number of available dedicated queues on a MPC drops below 10%. When the maximum number of dedicated queues is reached, a system log message, `COSD_OUT_OF_DEDICATED_QUEUES`, is generated. When the queue limit is reached, the system does not provide subsequent subscriber interfaces with a dedicated set of queues.

If the queue maximum is reached in a per unit scheduling configuration, new users get no queues as there are simply none left to assign. In contrast, with a hierarchical scheduling configuration, you can define remaining traffic profiles that can be used when the maximum number of dedicated queues is reached on the module. Traffic from all affected IFLs is then sent over a shared set of queues according to the traffic parameters that define the remaining profile.

> **Why Restricted Queues Aren't Needed on Trio**
>
> Defining restricted queues at the `[edit class-of-service restricted-queues]` hierarchy is never necessary on Trio MPCs. The restricted queue feature is intended to support a CoS configuration that references more than four FCs/queues on hardware that supports only four queues. The feature is not needed on Trio as all interfaces support eight queues.

Trio versus I-Chip/ADPC CoS Differences

Table 5-4 highlights key CoS processing differences between the older IQ2-based ADPC-based line cards and the newer Trio-based MPCs.

Table 5-4. ADPC (IQ2) and MPC CoS Compare and Contrast.

Feature	DPC-non-Q	DPCE-Q	MPC
Packet Granularity	64 B	512 B	128 B
Default Buffer	100 ms	500 ms	100 ms port based 500 ms for Q/EQ
Buffer Configured	Minimum	Maximum	Maximum
WRED	Head, with Tail assist	Tail Drop	Tail Drop
Port Level Shaping	NA	Supported	Supported
Queue Level Shaping	Single Rate	NA	Dual Rate per Queue
Egress mcast filtering	NA	NA	Supported
Egress Filter	Match on ingress protocol	Match on ingress protocol	Match on egress protocol
Overhead Accounting	Layer 2	Layer 2	Layer 1

Some key CoS differences between Trio-based Ethernet MPC and the I-chip-based DPCs include the following:

> A buffer configured on a 3D MPC queue is treated as the maximum. However, it is treated as the minimum on an I-chip DPC. On port-queuing I-chip DPCs, 64 byte-per-unit dynamic buffers are available per queue. If a queue is using more than its allocated bandwidth share due to excess bandwidth left over from other queues, its buffers are dynamically increased. This is feasible because the I-chip DPCs primarily perform WRED drops at the head of the queue, as opposed to "tail-assisted" drops, which are performed only when a temporal buffer is configured or when the queue becomes full. When a temporal buffer is not configured, the allocated

buffer is treated as the minimum for that queue and can expand if other queues are not using their share.

The Junos Trio chipset (3D MPC) maintains packets in 128-byte chunks for processing operations such as queuing, dequeuing, and other memory operations. J-Cell size over the fabric remains at 64 B.

Port shaping is supported on all MPCs.

Queues can have unique shaping and guaranteed rate configuration.

On MPCs with the Junos Trio chipset, WRED drops are performed at the tail of the queue. The packet buffer is organized into 128-byte units. Before a packet is queued, buffer and WRED checks are performed, and the decision to drop is made at this time. Once a packet is queued, it is not dropped. As a result, dynamic buffer allocation is not supported. Once the allocated buffer becomes full, subsequent packets are dropped until space is available, even if other queues are idle.

To provide larger buffers on Junos Trio chipset Packet Forwarding Engines, the delay buffer can be increased from the default 100 ms to 200 ms of the port speed and can also be oversubscribed using the `delay-buffer-rate` configuration on a per port. ADPC line cards base their shaping and queue statistics on Layer 2, which for untagged Ethernet equates to an additional 18 bytes of overhead per packet (MAC addresses, Type code, and FCS). In contrast, Trio chip sets compute queue statistics on Layer 1, which for Ethernet equates to an additional 20 bytes in the form of the 8 byte preamble and the 12 byte inter-packet gap. Note that Trio RED drop statistics are based on Layer 2 as the Layer 1 overhead is not part of the frame and is therefore not stored in any buffer.

Trio CoS Flow

Note that general packet processing was detailed in Chapter 1. In this section, the focus is on the specific CoS processing steps as transit traffic makes its way through a MX router's PFE complex. Figure 5-4 shows the major CoS processing blocks that *might* be present in a Trio PFE.

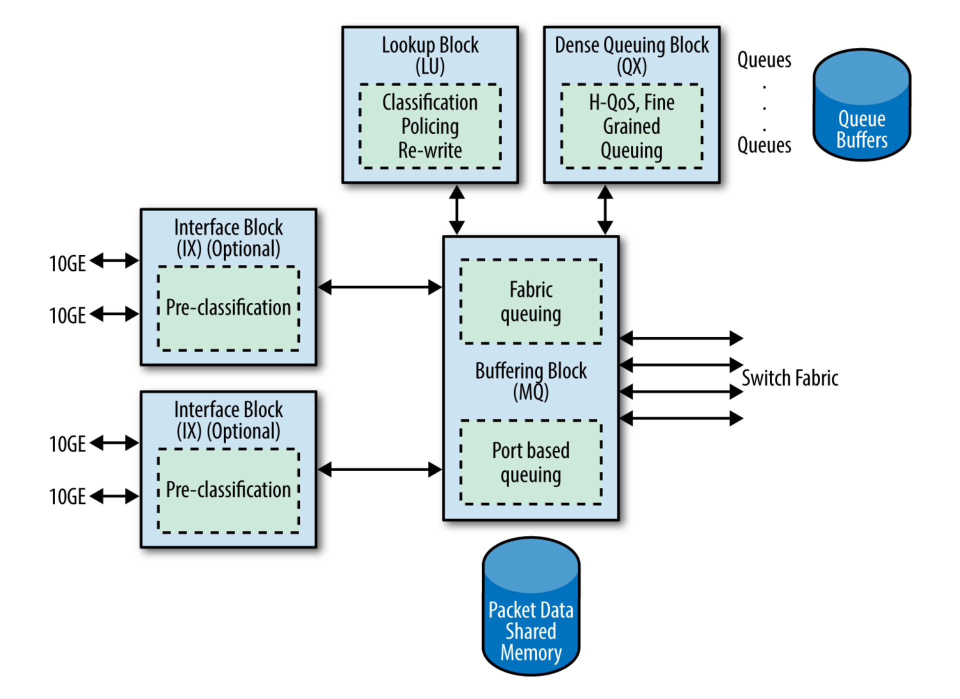


Figure 5-4. Trio PFE CoS Processing Points.

The flexibility of the Trio chipset means that not all MPCs have all processing stages shown (most MPCs do not use the Interface Block/IX stage, for example, and in many cases a given function such as preclassification or shaping can be performed in more than one location). While the details of the figure may not hold true in all present or future cases Trio PFE cases, it does represent the how the Trio design was laid out with regards to CoS processing. Future versions of the Trio PFE may delegate functions differently and may use different ASIC names, etc. The names were exposed here to provide concrete examples based on currently shipping Trio chipsets.

Starting with ingress traffic at the upper left, the first processing stage is preclassification, a function that can be performed by the interface block (IX) when present or by the Buffering Block (MQ) on line cards that don't use the IX.

Intelligent Oversubscription

The goal of preclassification is to prioritize network control traffic that is received over WAN ports (i.e., network ports, as opposed to switch fabric ports) into one of two traffic classes: network control and best effort. This classification is *independent* of any additional CoS processing that may be configured; here, the classification is based on a combination of the traffic's MAC address *and* deep packet inspection, which on Trio can go more than 256 bytes into the packet's payload. The independence of preclassification and conventional CoS classification can be clarified with the case of VoIP bearer traffic. Given that a bearer channel has only the media content and no signaling payload, such a packet is preclassified as best-effort at ingress from a WAN port. Given the need to prioritize voice for delay reasons, this same traffic will later be classified as EF (using either a BA or MF classifier), and then be assigned to a high-priority scheduler, perhaps with high switch fabric priority as well.

> Traffic received from switch fabric ports does not undergo preclassification, as that function should have been performed at ingress to the remote PFE. Switch fabric traffic includes traffic generated by the local host itself, which is therefore not preclassified.

Preclassification is performed on all recognized control protocols, whether the packet is destined for the local RE or for a remote host. The result is that control protocols are marked as high priority while noncontrol gets best effort or low-priority. Trio's fabric CoS then kicks in to ensure vital control traffic is delivered, even during times of PFE oversubscription, hence the term *intelligent oversubscription*.

The list of network control traffic that is recognized as part of preclassification in the 11.4R1 release can be found at *http://www.juniper.net/techpubs/en_US/junos11.4/topics/concept/trio-mpc-mic-intelligent-oversub-overview-cos-config-guide.html*.

The preclassification feature is not user-configurable. As of v11.4, the list of protocols includes the following:

Layer 2:

ARPs: Ethertype 0x0806 for ARP and 0x8035 for dynamic RARP

IEEE 802.3ad Link Aggregation Control Protocol (LACP): Ethertype 0x8809 and 0x01 or 0x02 (subtype) in first data byte

IEEE 802.1ah: Ethertype 0x8809 and subtype 0x03

IEEE 802.1g: Destination MAC address 0x01–80–C2–00–00–02 with Logical Link Control (LLC) 0xAAAA03 and Ethertype 0x08902

PVST: Destination MAC address 0x01–00–0C–CC–CC–CD with LLC 0xAAAA03 and Ethertype 0x010B 382

xSTP: Destination MAC address 0x01–80–C2–00–00–00 with LLC 0x424203

GVRP: Destination MAC address 0x01–80–C2–00–00–21 with LLC 0x424203

GMRP: Destination MAC address 0x01–80–C2–00–00–20 with LLC 0x424203

IEEE 802.1x: Destination MAC address 0x01–80–C2–00–00–03 with LLC0x424203

Any per-port my-MAC destination MAC address

Any configured global Integrated Bridging and Routing (IRB) my-MAC destination MAC address

Any PPP encapsulation (Ethertype 0x8863 [PPPoE Discovery] or 0x8864 [PPPoE Session Control]) is assigned to the network control traffic class (queue 3).

Layer 3 and Layer 4:

IGMP query and report: Ethertype 0x0800 and carrying an IPv4 protocol or IPv6next header field set to 2 (IGMP)

IGMP DVRMP: IGMP field version = 1 and type = 3

IPv4 ICMP: Ethertype 0x0800 and IPv4 protocol = 1 (ICMP)

IPv6 ICMP: Ethertype 0x86DD and IPv6 next header field = 0x3A (ICMP)

IPv4 or IPv6 OSPF: Ethertype 0x0800 and IPv4 protocol field or IPv6 next headerfield = 89 (OSPF)

IPv4 or IPv6 VRRP: IPv4 Ethertype 0x0800 or IPv6 Ethertype 0x86DD and IPv4protocol field or IPv6 next header field = 112 (VRRP)

IPv4 or IPv6 RSVP: IPv4 Ethertype 0x0800 or IPv6 Ethertype 0x86DD and IPv4protocol field or IPv6 next header field = 46 or 134

IPv4 or IPv6 PIM: IPv4 Ethertype 0x0800 or IPv6 Ethertype 0x86DD and IPv4protocol field or IPv6 next header field = 103

IPv4 or IPv6 IS-IS: IPv4 Ethertype 0x0800 or IPv6 Ethertype 0x86DD and IPv4 protocol field or IPv6 next header field = 124

IPv4 router alert: IPv4 Ethertype 0x0800 and IPv4 option field = 0x94 (the RA option itself is coded as a decimal 20, but other bits such as length and the class/copy flags are also present)

IPv4 and IPv6 BGP: IPv4 Ethertype 0x0800 or IPv6 Ethertype 0x86DD, TCP port = 179, and carrying an IPv4 protocol or IPv6 next header field set to 6 (TCP)

IPv4 and IPv6 LDP: IPv4 Ethertype 0x0800 or IPv6 Ethertype 0x86DD, TCP orUDP port = 646, and carrying an IPv4 protocol or IPv6 next header field set to6 (TCP) or 17 (UDP)

IPv4 UDP/L2TP control frames: IPv4 Ethertype 0x0800, UDP port = 1701, andcarrying an IPv4 protocol field set to 17 (UDP)

DHCP: Ethertype 0x0800, IPv4 protocol field set to 17 (UDP), and UDP destinationport = 67 (DHCP service) or 68 (DHCP host)

IPv4 or IPv6 UDP/BFD: Ethertype 0x0800, UDP port = 3784, and IPv4 protocolfield or IPv6 next header field set to 17 (UDP)

The Remaining CoS Packet Flow

The packet is then spread into shared memory by the Buffer Block (MQ) stage, while the notification cell is directed to the route lookup function, a function provided by the Lookup Block (LU). This stage also performs classification (BA or filter-based multifield), as well as any policing and packet header rewrite functions. The memory manager handles the queuing of traffic for transmission over the switch fabric to a remote PFE, and can provide port-based shaping services when the process is not offloaded to the QX ASIC, or when the QX ASIC is not present, thereby making the line card capable of port-based queuing only.

The queue management ASIC (QX) is only present on MPCs that offer fine-grained queuing and H-CoS, as noted by their Q or EQ designation. When present, the queue management stage handles scheduling and queuing at the port, IFL, IFL-Set, and queue levels, for a total of four levels of scheduling and queuing hierarchy in the current H-CoS offering.

CoS Processing: Port- and Queue-Based MPCs

With general Trio CoS processing covered, things move to Figure 5-5, which shows the CoS touch points for packets flowing though a queuing MPC.

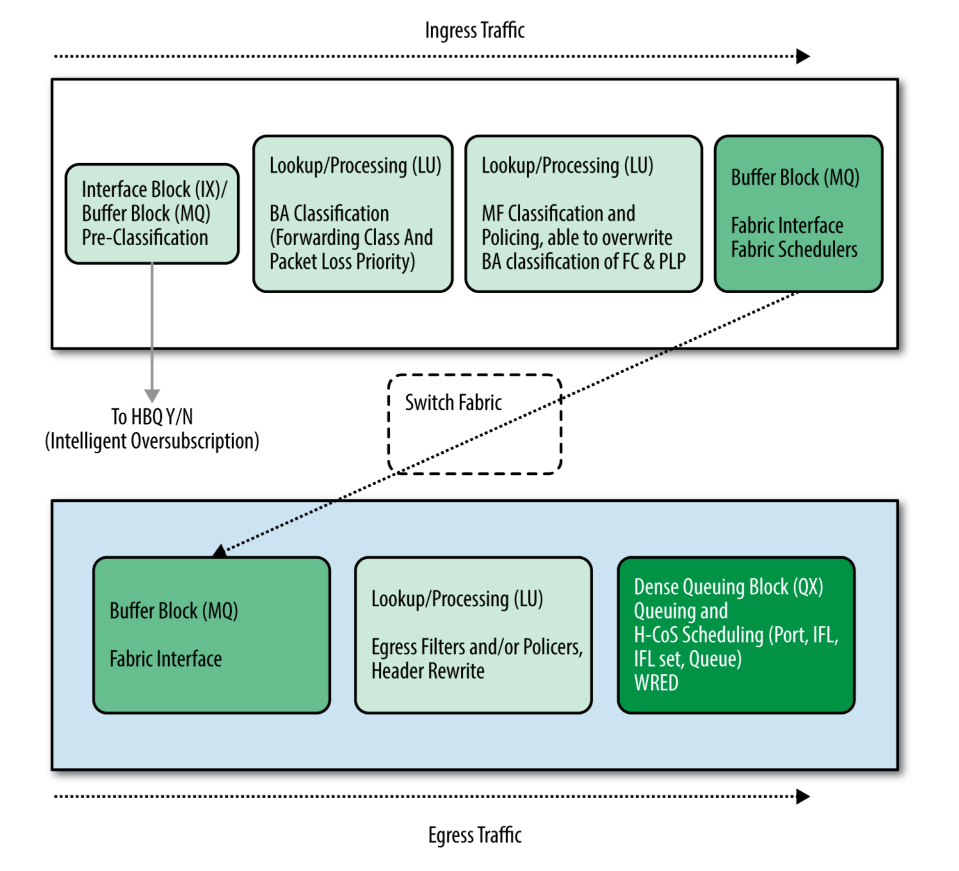


Figure 5-5. Port and Queuing MPC CoS Flow.

Here, the flow is shown in a more conventional left-to-right manner with the switch fabric between the ingress PFE on the top and the egress PFE on the bottom. It's worth noting how some chips appear more than once in the flow, stepping in to perform a different CoS function at various points in the processing chain.

In Figure 5-5, things begin at the upper left, where traffic ingresses into the PFE on a *WAN* facing port, which is to say a MIC interface of some kind, which is somewhat ironic given it's likely to be Ethernet-based, which is of course a *LAN* technology. The term WAN is chosen to contrast with traffic that can ingress into the PFE from the switch fabric. Besides, you can buy SONET-based MICs that provide channelized services with Frame Relay and PPP support, in which case the stream truly does arrive from a WAN port.

Switch Fabric Priority

As noted previously, traffic arriving from WAN ports is subjected to a preclassification function, which can be performed at a variety of locations depending on hardware specifics of a given MPC or MIC type. Currently, preclassification supports only two classes: best effort and control. Transit traffic is an example of best effort, whereas control traffic such as ARP or OSPF that is either destined to the local or a remote host, is marked with the higher priority.

The preclassification function causes a congested PFE to drop low-priority packets first to help keep the control plane stable during periods of congestion that stem from PFE oversubscription. Oversubscription can occur when many flows that ingress on multiple PFEs converging to egress on a single egress PFE. The preclassification function provides Trio MPCs with an intelligent oversubscription capability that requires no manual intervention or configuration.

In addition to pre-classification, you can map traffic to one of two switch fabric priorities, high or low, and if desired even link to one or more WRED drop profiles to help tailor drop behavior, as described in a later section.

Classification and Policing

The next step has the Lookup Processing chip (LU) performing Behavior Aggregate (BA) classification to assign the packet to a forwarding class (FC) and a loss priority. The type of BA that is performed is a function of port mode and configuration. Layer 2 ports typically classify on Ethernet level 802.1p or MPLS EXP/TC bits, while Layer 3 ports can use DSCP, IP precedence, or 802.1p. Fixed classification, where all traffic received on a given IFL is mapped to a fixed FC and loss priority, is also supported.

At the next stage, the LU chip executes any ingress filters, which can match upon multiple fields in Layer 2 or Layer 3 traffic and perform actions such as changing the FC, loss priority, or evocation of a policer to rate limit the traffic. As shown in Figure 5-5, MF classifier occurs after BA classification, and as such it can overwrite the packet's FC and loss priority settings.

Traffic then enters the Buffer Manger chip (MQ), assuming of course that no discard action was encountered in the previous input filter processing stage. The route lookup processing that is also performed in the previous stage will have identified one or more egress FPCs (the latter being the case for broadcast/multicast traffic) to which the traffic must be sent. The MQ chip uses a request grant arbitration scheme to access the switch fabric in order to send J-cells over the fabric to the destination FPC. As noted previously, the Trio switch fabric supports a priority mechanism to ensure critical traffic is sent during periods of congestions, as described in the following.

Classification and Rewrite on IRB Interfaces. Integrated Bridging and Routing (IRB) interfaces are used to tie together Layer 2 switched and Layer 3 routed domains on MX routers. MX routers support classifiers and rewrite rules on IRB interface at the `[edit class-`

`of-service interfaces irb unit logical-unit-number]` level of the hierarchy. All types of classifiers and rewrite rules are allowed, including IEEE 802.1p.

The IRB classifiers and rewrite rules are used only for "routed" packets; in other words, it's for traffic that originated in the Layer 2 domain and is then routed through the IRB into the Layer 3 domain, or vice versa. Only IEEE classifiers and IEEE rewrite rules are allowed for pure Layer 2 interfaces within a bridge domain.

Egress Processing

Egress traffic is received over the fabric ports at the destination PFE. The first stage has the packet chunks being received over the fabric using the request grant mechanism, where the 64-byte J-cells are again stored in shared memory, only now on the egress line card. The notification cell is then subjected to any output filters and policers, a function that is performed by the Lookup and Processing (LU) ASIC, again now on the egress PFE.

Egress Queuing: Port or Dense Capable? The majority of CoS processing steps and capabilities are unchanged between regular and queuing MPCs. The final step in egress CoS processing involves the actual queuing and scheduling of user traffic. Dense queuing MPCs offer the benefits of per unit or H-CoS scheduling, as described in a later section. Both of these modes allow you to provide a set of queues to individual IFLs (logical interfaces, or VLANs/subscribers). H-CoS extends this principal to allow high levels of queue and scheduler node scaling over large numbers of subscribers.

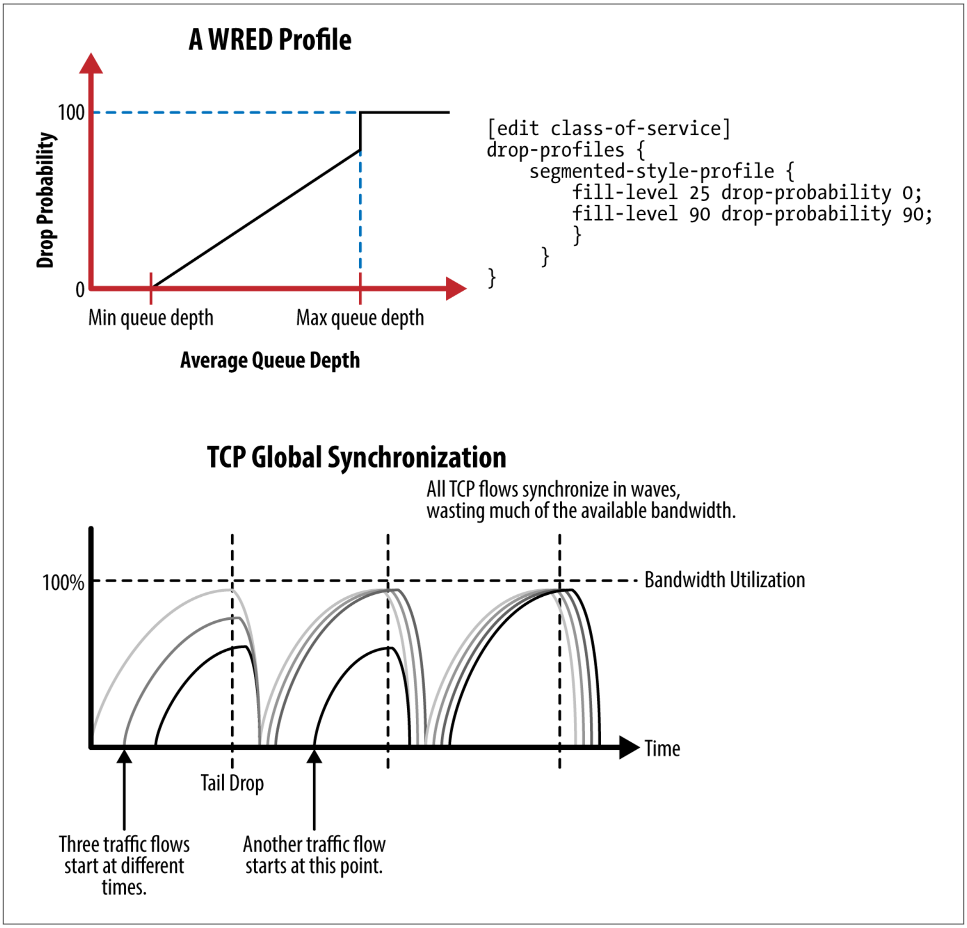
Port-based MPCs currently support port-level shaping and scheduling over a set of shared queues only. Port mode scheduling is performed by the Buffer Block (MQ) ASIC on nonqueuing MPCs. On queuing MPCs, the dense queuing block handles port, per unit, and hierarchical scheduling modes.

As of the v11.4R1 release, ingress queuing is not supported. Future releases may offer the ability to perform H-CoS at ingress, in which case the functionality describe here for egress queuing is expected to be available at ingress. Ingress queuing can be useful in certain situations where ingress to egress traffic is so unbalanced that congestion occurs on the egress FPC. Ingress queuing moves that congestion closer to the source, specifically to the ingress PFE, where WRED discards can relieve switch fabric burden for traffic that was likely to be discarded on the egress FPC anyway.

The final egress processing stage also performs WRED congestion management according to configured drop profiles to help prevent congestion before it becomes so severe that uncontrolled tail drops must be performed once the notification queue fills to capacity.

WRED. WRED is used to detect and prevent congestion by discarding packets based on queue fill levels, with the expectation that the senders are TCP-based and sense loss as an indication of congestion, at which point the TCP window is reduced to affect flow control on the endpoints. The primary goal of a WRED algorithm is to affect implicit flow control when impending congestion is sensed to try and avoid a phenomena known as global TCP synchronization. Global synchronization occurs when TCP sources ramp up their transmit rates in unison until buffers are exhausted, forcing uncontrolled trail drops, which in turn causes all senders to back down, again in unison, resulting in inefficient bandwidth utilization.

WRED does not work well for non-TCP sources, which are typical transport choices for real-time applications and often based on UDP; a protocol does not support congestion windows or retransmissions. Figure 5-6 shows a WRED drop profile, as well as a graphic showing the concept of TCP global synchronization.


A WRED Profile
100
Drop Probability
0
Min queue depth
Max queue depth
Average Queue Depth
TCP Global Synchronization
All TCP flows synchronize in waves,
wasting much of the available bandwidth.
100%
Bandwidth Utilization
Time
Tail Drop
Three traffic flows start at different times.
Another traffic flow starts at this point.

```
[edit class-of-service]
drop-profiles {
    segmented-style-profile {
        fill-level 25 drop-probability 0;
        fill-level 90 drop-probability 90;
        }
     }
}
```

Figure 5-6. A WRED Profile and the Issue of TCP Global Synchronization

The figure shows a sample WRED profile that is set with a minimum and maximum queue fill level, along with a corresponding drop probability. Based on the configuration, no WRED drops occur until the queue reaches an average 25% fill, at which point drop probability increases as a function of fill until you hit 100% at the max fill level, which in this case is 90%.

In Trio, WRED drop actions occur *before* a notification cell is enqueued, which is to say that Trio WRED performs drops at the tail of the queue. A WRED-based drop at the tail of a queue is not the same thing as a tail drop, as the latter signifies uncontrolled drop behavior due to a buffer being at capacity. In Trio, a given packet is subject to WRED drops based on the configured drop profile as compared to *average queue depth*, as well as to tail drops that are based on *instantaneous queue depth*. The benefit here is that even if a buffer is at capacity, you still get WRED-based intelligent drop behavior given that WRED is being performed at the tail, prior to the packet needing any buffer, as it has not been queued yet.

Given that tail drops can be seen as a bad thing, a bit more on the Trio tail drop-based WRED algorithm is warranted here. When there is no buffer to enqueue a packet, it will be tail dropped initially. However, if the traffic rate is steady, the drops transition from tail to WRED drops. Given that WRED works on average queue lengths, under chronic congestion the average will catch up with the instant queue depth, at which point the drops become WRED-based.

Once a packet is queued, it is not dropped. As a result, dynamic buffer allocation is not supported on Trio. The buffer that is allocated to each queue (based on configuration) is considered the maximum for that queue. Once the allocated buffer becomes full, subsequent packets are dropped until space is available, even if other queues are idle.

Enhanced queuing (EQ) Trio MPC/MIC interfaces support up to 255 drop profiles, up to 128 tail-drop priorities for guaranteed low (GL) priorities, and 64 each for guaranteed high and medium priorities. You can have up to four WRED profiles in effect per queue.

Currently, Trio does not support protocol-based WRED profiles. That is, you must use `protocol any` as opposed to `protocol tcp`, with the latter returning a commit error. This means cannot link to different WRED profiles based on protocol such as TCP versus non-TCP.

Trio Hashing and Load Balancing

Hashing is a function by which various fields in a packet are used to match on a flow, which is a sequence of packets between the same source and destination addresses and ports, and then map that flow to one of several equal cost links. Junos supports Equal Cost Multi-Path (ECMP) load balancing over as many as 16 paths and offers the default per prefix as well as the (poorly named) `per-packet` option, which really means per flow. In addition to ECMP, Junos also supports hashing to balance traffic over member links that are part of an AE bundle. In v11.4, an AE bundle can contain as many as 16 member links.

Hashing and load balancing is one of those areas where it's almost impossible to please everyone all the time. The hash that yields near perfect balance in one network may result in unbalanced traffic on another due to the simple fact that different traffic encapsulations, combined with differing hash capabilities, hashing defaults, and configuration specifics, add up to yield so many permutations.

Trio-based PFEs had the advantage of being developed years after the original M and even T-series platforms were rolled into production networks. Juniper engineering listened to customer feedback and started Trio with a clean hashing slate in order to break from previous limitations and shortcomings. The result is that Trio offers some unique hash and load balancing capabilities when compared to other Junos devices.

In fact, Trio LB is such a departure from historic Junos platforms that a whole new area of configuration hierarchy was created just to support it.

In the v11.4 release testing, it was found that Trio platforms don't always ignore legacy hash configuration found at the [edit forwarding-options hash-key] hierarchy, which can lead to unpredictable operation. Be sure to leave the legacy hash-key stanza empty on a Trio-based MX router.

Trio hashing functionality is configured using the enhanced-hash-key keyword under forwarding-options:

```
{master}[edit]
jnpr@R1-RE0# set forwarding-options enhanced-hash-key ?
Possible completions:
+ apply-groups         Groups from which to inherit configuration data
+ apply-groups-except  Don't inherit configuration data from these groups
> family               Protocol family
> services-loadbalancing  Select key to load balance across service PICs
{master}[edit]
jnpr@R1-RE0# set forwarding-options enhanced-hash-key
```

The services-load balancing load balancing statement, as its name implies, controls how traffic that needs services applied is balanced when a system has multiple service engines (i.e., MS-DPCs). It distributes traffic across the services PICs based on source IP address when a route pointing to more than one services PICs is installed. This option is mandatory to provide stateful services such as Carrier Grade NAT. In fact, by using source IP only based load balancing, all sessions coming from the same source will be load balanced to the same NPU for session state tracking. As of v11.4, the options for services load balancing include incoming interface and source IP address:

```
{master}[edit]
jnpr@R1-RE0# set forwarding-options enhanced-hash-key services-loadbalancing family
              inet layer-3-services ?
Possible completions:
```

```
+ apply-groups         Groups from which to inherit configuration data
+ apply-groups-except  Don't inherit configuration data from these groups
  incoming-interface-index  Include incoming interface index in the hash key
  source-address       Include IP source address in the hash key
```

The options for transit traffic load balancing vary by protocol family. For IPv4, the CLI offers the following options:

```
{master}[edit]
jnpr@R1-RE0# set forwarding-options enhanced-hash-key family inet ?
Possible completions:
+ apply-groups         Groups from which to inherit configuration data
+ apply-groups-except  Don't inherit configuration data from these groups
  incoming-interface-index  Include incoming interface index in the hash key
  no-destination-port  Omit IP destination port in the hash key
  no-source-port       Omit IP source port in the hash key
  type-of-service      Include TOS byte in the hash key
```

Table 5-5 shows all IPv4 fields that are factored by the enhanced hash key, if they are configurable, and whether they are present in a default configuration.

Table 5-5. IPv4 Enhanced Hash Fields.

IPv4 Field	Configurable	Default
Incoming interface	Yes	No
Destination IP	No	Yes
Source IP	No	Yes
Protocol ID	No	Yes
Source/Destination Ports	Yes	Yes
DSCP	Yes	No

Table 5-6 does the same, but now for IPv6:

Table 5-6. IPv6 Enhanced Hash Fields.

IPv6 Field	Configurable	Default
Incoming interface	Yes	No
Destination IP	No	Yes
Source IP	No	Yes
Protocol ID	No	Yes
Source/Dest Ports	Yes	Yes
Traffic Class	Yes	No

And now, for the multiservice family, which includes bridged, CCC, and VPLS protocols in Table 5-7.

Table 5-7. Multiservice Enhanced Hash Fields.

Multiservice (L2) field	Configurable	Default
Incoming interface	Yes	No
Destination MAC	No	Yes
Source MAC	No	Yes
Outer 802.1p	Yes	No
Ethertype/Payload	Yes	Yes
Payload IPv4/IPv6	See IPv4/V6 above	See IPv4/V6 above

And last, but not least, the MPLS family in Table 5-8.

Table 5-8. MPLS Enhanced Hash Fields.

MPLS Field	Configurable	Default
Incoming interface	Yes	No
Top 5 labels	No	Yes
Outermost Label EXP (TC)	Yes	No
Payload*	Yes	Yes

As shown in Table 5-8, the enhanced hash key for MPLS can factor the payload. An interesting feat, given that MPLS lacks an explicit protocol type field so such information is not explicitly available at the MPLS layer. Once again, the Trio PFE's ability to peer deep into a packet's payload is leveraged to try and do the right thing, using the following sequence:

> First, IPv4 is assumed if the first nibble following the bottom label is 0x4. If found, the fields in the table on IPv4 are used to load-balance the payload.
> Else, IPv6 is assumed if the first nibble following the bottom label is 0x6. If found, the fields in the table on IPv6 are used to load-balance the payload.
> Else, Ethernet is assumed and the chipset attempts to parse the remaining bytes as an Ethernet header using the information in the table for Multiservice. As part of this process, the EtherType in the presumed Ethernet header is examined, and if found to be IP, the identified Layer 3 payload (IPv4 or IPv6) is then factored into the load-balancing hash.

As an indication of Trio flexibility, MPLS enhanced hash supports the identification of an IP layer even if the MPLS payload contains an Ethernet frame with two VLAN tags! Indeed, that's pretty impressive at 100G Ethernet speeds!

ISO CNLP/CNLS Hashing and Load Balancing

Though apparently not well documented, Trio-based platforms offer support for ISO's Connectionless Network Layer Service (CNLS), which is based on routing the Connectionless Network Layer Protocol (CNLP), which in turn makes use of the IS-IS and ES-IS routing protocols. For CNLP traffic, the input hash parameters are the source and destination NSAP addresses to provide symmetrical load balancing.

You can view the current load balancing settings using the following VTY command on the desired FPC. The following represents the defaults:

```
NPC3(router vty)# show jnh lb
Unilist Seed Configured 0x0145db91 System Mac address 00:24:dc:77:90:00
Hash Key Configuration: 0x0000000000e00000 0xffffffffffffffff
           IIF-V4: No
         SPORT-V4: Yes
         DPORT-V4: Yes
              TOS: No

           IIF-V6: No
         SPORT-V6: Yes
         DPORT-V6: Yes
    TRAFFIC_CLASS: No

         IIF-MPLS: No
     MPLS_PAYLOAD: Yes
         MPLS_EXP: No

      IIF-BRIDGED: No
    MAC ADDRESSES: Yes
    ETHER_PAYLOAD: Yes
     802.1P OUTER: No

Services Hash Key Configuration:
         SADDR-V4: No
           IIF-V4: No
```

A Forwarding Table Per-Packet Policy Is Needed

In all cases, for load balancing to take effect you must apply a `per-packet` load-balancing policy to the forwarding table; by default, Junos does `per-prefix` load balancing. This policy allows the PFE to install multiple forwarding next hops for a given prefix. Despite the name, this is actual per flow, hashed according to the previous descriptions based on protocol type. A typical per packet load-balancing policy is shown:

```
{master}[edit]
jnpr@R1-RE0# show routing-options forwarding-table
export lb-per-flow;


{master}[edit]
jnpr@R1-RE0# show policy-options policy-statement lb-per-flow
term 1 {
    then {
        load-balance per-packet;
        accept;
        }
    }
```

Load Balancing and Symmetry

By default on nonaggregated interfaces, the Trio hashing function is designed to result in symmetric flows. This is to say that if MX router A hashes a flow to a given link, than its (Trio-based MX) neighbor will in turn hash the matching return traffic to that same link. You can tune hashing parameters to force asymmetry when desired.

Aggregate Ethernet interfaces try to randomize their hashes to add a perturbance factor so that the same hash decision is not made a set of cascaded nodes all using AE interfaces. Without the randomization, a polarization can occur, which is when all nodes hash the same flow to the same AE link member throughout the entire chain.

The randomization solves the polarization issue, but this comes at the cost of breaking the default symmetry of Trio load balancing. You can use the following options to work around this when symmetric AE load balancing is more important than randomizing the hash at each node. Note that you need the same link index hash specified at both ends of the link, and that the `symmetric` option is currently hidden in the v11.4R1 release. While supported, the use of any hidden option should only be used in a production network after consultation with Juniper TAC:

```
{master}[edit]
jnpr@R1-RE0# set forwarding-options enhanced-hash-key symmetric
```

You must be sure to set the same link index for all child IFLs in the AE bundle as well:

```
{master}[edit]
jnpr@R1-RE0# set interfaces xe-2/0/0 gigether-options 802.3ad link-index ?
Possible completions:
<link-index>Desired child link index within the Aggregated Interface (0..15)
{master}[edit]
```

Key Aspects of the Trio CoS Model

Trio PFEs handle CoS differently than previous IQ2 or IQ2E cards. Some key points to note about the Trio queuing and scheduling model include the following.

Independent Guaranteed Bandwidth and Weight

The model separates the guaranteed bandwidth concept from the weight of an interface node. Although often used interchangeably, guaranteed bandwidth is the bandwidth a node can use when it wants to, independently of what is happening at the other nodes of the scheduling hierarchy. On the other hand, the weight of a node is a quantity that determines how any excess bandwidth is shared.

The weight is important when the siblings of a node (that is, other nodes at the same level) use less than the sum of their guaranteed bandwidths. In some applications, such as constant bit rate voice where there is little concern about excess bandwidth, the guaranteed bandwidth dominates the node; whereas in others, such as bursty data, where a well-defined bandwidth is not always possible, the concept of weight dominates the node. As an example, consider the Peak Information Rate (PIR) mode where a Guaranteed Rate (G-Rate) is not explicitly set, combined with queues that have low transmit rate values. In this case, a queue may be above its transmit rate, using excess bandwidth most of the time. In contrast, in the Committed Information Rate (CIR) mode, where CIR = G-Rate, the gap between the G-Rate and shaping rate tends to determine how big a role excess rate weighting has on a queue's bandwidth. Details on the PIR and CIR modes of operation are provided in the Hierarchical CoS section.

Guaranteed versus Excess Bandwidth and Priority Handling

The model allows multiple levels of priority to be combined with guaranteed bandwidth in a general and useful way. There is a set of three priorities for guaranteed levels and a set of two priorities for excess levels that are at a lower absolute level. For each guaranteed level, there is only one excess level paired with it. You can configure one guaranteed priority and one excess priority. For example, you can configure a queue for guaranteed low (GL) as the guaranteed priority and configure excess high (EH) as the excess priority.

Nodes maintain their guaranteed priority level for GH and GM traffic. If the node runs low on G-Rate, it demotes GL into excess to keep the guaranteed path from being blocked. When performing per-priority shaping, the node reduces the priority of traffic in excess of the shaper, except for EL, which is already at the lowest priority.

When demoting, the source queue's settings control the value of the demotion. A queue set to excess none therefore blocks demotion at scheduler nodes.

If the queue bandwidth exceeds the guaranteed rate, then the priority drops to the excess priority (for example, excess high [EH]). Because excess-level priorities are lower than their guaranteed counterparts, the bandwidth guarantees for each of the other levels can be maintained.

Trio MPC/MIC interfaces do not support the `excess-bandwidth-sharing` statement. You can use the `excess-rate` statement in scheduler maps and traffic control profiles instead.

Input Queuing on Trio

Currently, input queuing is not supported on the Trio MPC/MIC interfaces. This capability is totally supported by the hardware but not yet implemented in JUNOS as of the v11.4 release. This capability is expected in an upcoming Junos release, but there is a price to pay, which in this case is a reduction of the PFE's bandwidth capacity by one-half. This is because to enable simultaneous ingress/egress class of service processing, the CoS engine has to process the traffic twice, hence halving its total throughput. At this time, it's expected that all egress CoS features will be also provided in ingress with the following known exceptions:

Classification for ingress queuing will be only available using BA classifiers.
Interface-set (Level 2 nodes) won't be supported in ingress.
Aggregated Ethernet won't be supported.

Trio Buffering

The Trio MPC/MIC interfaces do not support the `q-pic-large-buffer` statement at the`[edit chassis fpc fpc-number pic pic-number]` hierarchy level. All tunnel interfaces have 100-ms buffers. The `huge-buffer-temporal` statement is not supported.

In most cases, MPCs provide 100 ms worth of buffer per port when the delay buffer rate is 1 Gbps or more, and up to 500 ms worth of buffer when the delay buffer rate is less than 1 Gbps. The maximum supported value for the delay buffer is 256 MB and the minimum value is 4 kB, but these values can vary by line card type. In addition, due to the limited number of drop profiles supported and the large range of supported speeds, there can be differences between the user-configured value and the observed hardware value. In these cases, hardware can round the configured values up or down to find the closest matching value.

When the Trio MPC/MIC interface's delay buffers are oversubscribed by configuration (that is, the user has configured more delay-buffer memory than the system can support), the configured WRED profiles are implicitly scaled down to drop packets more aggressively from the relatively full queues. This creates buffer space for packets in the relatively empty queues and provides a sense of fairness among the delay buffers. There is no configuration needed for this feature.

Trio Drop Profiles

The enhanced queuing (EQ) Trio MPC/MIC interfaces support up to 255 drop profiles and up to 128 tail-drop priorities for guaranteed low (GL) priorities and 64 each for guaranteed high and medium priorities. Dropping due to congestion is done by making two decisions: first a WRED decision is made, and then a tail drop decision is made. The time averaged queue length represents level of congestion of the queue used by the WRED drop decision. The instantaneous queue length represents the level of congestion of the queue used by the tail drop decision.

Trio Bandwidth Accounting

Trio MPC/MIC interfaces take all Layer 1 and Layer 2 overhead bytes into account for all levels of the hierarchy, including preamble, interpacket gaps, and the frame check sequence (cyclic redundancy check). Queue statistics also take these overheads into account when displaying byte statistics; note that rate limit drop byte counts reflect only the frame overhead and don't include any preamble or IPG bytes (18 vs. 38 Bytes). On I-chip/IQ2, Layer 3 interface statistics are based on the Layer 3 rate, but the shaping itself is based on Layer 2 rate; IQ2Bridge interfaces display statistics based on Layer 2 rate.

Trio MPCs allow you to control how much overhead to count with the `traffic-manager` statement and its related options. By default, an overhead of 24 bytes (20 bytes for the header, plus 4 bytes of CRC) is added to egress shaping statistics. You can configure the system to adjust the number of bytes to add or subtract from the packet when shaping. Up to 124 additional bytes of overhead can be added or up to120 bytes can be subtracted. As previously noted, Trio differs from IQ2 interfaces in that, by default, Trio factors Ethernet Layer 1 overhead, to include 20 bytes for the preamble and inter-frame gap, in addition to the 18 bytes of frame overhead, as used for MAC addresses, the type code, and the FCS. Thus, the default shaping overhead for Trio is 38 bytes per frame. Subtract 20 bytes to remove the preamble and IPG from the calculation, which in turn matches the Trio overhead and shaping calculations to those used by IQ2/IQ2E interfaces.

Trio Shaping Granularity

Trio MPC/MIC interfaces have a certain granularity in the application of configured shaping and delay buffer parameters. In other words, the values used are not necessarily precisely the values configured. Nevertheless, the derived values are as close to the configured values as allowed. For the Trio MPC, the shaping rate granularity is 250 kbps for coarse-grained queuing on the basic hardware and 24 kbps for fine-grained queuing on the enhanced queuing devices.

With hierarchical schedulers in oversubscribed PIR mode, the guaranteed rate for every logical interface unit is set to zero. This means that the queues transmit rates are always oversubscribed, which makes the following true:

> If the queue transmit rate is set as a percentage, then the guaranteed rate of the queue is set to zero, but the excess rate (weight) of the queue is set correctly.
>
> If the queue transmit rate is set as an absolute value and if the queue has guaranteed high or medium priority, then traffic up to the queue transmit rate is sent at that priority level. However, for guaranteed low traffic, that traffic is demoted to the excess low region. This means that best-effort traffic well within the queue transmit rate gets a lower priority than out-of-profile excess high traffic. This differs from the IQE and IQ2E PICs.

Trio MPLS EXP Classification and Rewrite Defaults

Trio PFEs do not have a default MPLS EXP classifier or rewrite rule in effect.

> RFC 5462 renames the MPLS EXP field to Traffic Class (TC); the functionality remains the same, however.

If your network's behavior aggregate (BA) classifier definitions do not include a custom EXP classifier and matching rewrite table, then you should at least specify the defaults using a `rewrite-rules exp default` statement at the `[edit class-of-service interfaces interface-name unit logical-unit-number]` hierarchy level. Doing so ensures that MPLS EXP value is rewritten according to the default BA classifier rules, which are based on forwarding class and packet loss priority being mapped into the EXP field. This is especially important for Trio PFEs, which unlike other M and T-series platforms *don't have a default EXP* classifier or rewrite rule in effect, which can cause unpredictable behavior for MPLS packets, such as having the IP TOS value written into the label. To illustrate, this is from a M120 with no CoS configuration:

```
[edit]
user@M120# show class-of-service

[edit]
user@M120#

user@M120# run show class-of-service interface ge-2/3/0
Physical interface: ge-2/3/0, Index: 137
Queues supported: 8, Queues in use: 4
  Scheduler map: <default>, Index: 2
  Input scheduler map: <default>, Index: 2
  Chassis scheduler map: <default-chassis>, Index: 4
  Congestion-notification: Disabled

  Logical interface: ge-2/3/0.0, Index: 268
    Object                  Name                   Type                    Index
    Rewrite                 exp-default            exp (mpls-any)             33
    Classifier              exp-default            exp                        10
    Classifier              ipprec-compatibility   ip                         13
```

Compared to a Trio-based MX, also with a factory default (no) CoS configuration:

```
{master}[edit]
jnpr@R1-RE0# show class-of-service

{master}[edit]
jnpr@R1-RE0#

jnpr@R1-RE0# run show class-of-service interface xe-2/1/1
Physical interface: xe-2/1/1, Index: 151
Queues supported: 8, Queues in use: 4
  Scheduler map: <default>, Index: 2
  Congestion-notification: Disabled

  Logical interface: xe-2/1/1.0, Index: 330
    Object                  Name                   Type                    Index
    Classifier              ipprec-compatibility   ip                         13
```

Trio CoS Processing Summary

Trio-based MX platforms offer flexible CoS capabilities at scale. Full support for Layer 3 IPv4 and IPv6 MF and BA classification as well as header rewrite is available, as is Layer 2-based equivalents using MPLS EXP, IEEE 802.1p, or IEEE 802.1ad (also known as QinQ) fields, including double label rewrite capability. The ability to perform MF classification on IP and transport levels fields for traffic that is within a Layer 2 bridged frame is a powerful indicator of the chipset's flexibility, in part enabled by the ability to peer up to 256 bytes into a packet!

Trio can back up the brute force of its interface speeds with sophisticated priority-based scheduling that offers control over excess bandwidth sharing, with the ability to scale to thousands of queues and subscribers per port, with as many as eight queues per subscriber to enable triple-play services today, and into the future. Rich control over how traffic is balanced helps make sure you get the most use out of all your links, even when part of an AE bundle.

Lastly, H-CoS, the lengthy subject of the next section, offers network operators the ability to support and manage large numbers of subscribers, as needed into today's Broadband Remote Access Server/Broadband Network Gateway (B-RAS/BNG)-based subscriber access networks, where the ability to offer per-user- as well user-aggregate-level CoS shaping is waiting to enable differentiated services, and increased revenue, within your network.

Hierarchical CoS

This section details Hierarchical CoS. Before the deep dive, let's get some terminology and basic concepts out of the way via Table 5-9 and its terminology definitions.

Table 5-9. H-CoS and MX Scheduling Terminology.

Term	Definition
CIR	Committed information rate, also known as "guaranteed rate." This parameters specifies the minimum bandwidth for an IFL-Set or VLAN/IFL.
C-VLAN	A Customer VLAN, the inner tag on a dual tagged frame, often used interchangeably with IFL as each customer VLAN is associated with a unique IFL. See also S-VLAN.
Excess-priority	Keyword in a class of service scheduler container. Specifies the priority of excess bandwidth. Excess bandwidth is the bandwidth available after all guaranteed-rates have been satisfied. Options are Excess High or Low (EH/EL).
Excess-rate	Keyword in class of service traffic control profile and scheduler containers. Specifies how excess bandwidth is distributed amongst peers in a scheduler-hierarchy.
Guaranteed-rate (G-Rate)	See "CIR." In the Trio Queuing Model, the guaranteed rate is denoted as "G" for guaranteed. G-Rate priority is Strict-High/High, Medium, or Low (SH, H, M, L). Queues transmit rate is considered a G-Rate when not overbooked, else the committed information rate in Traffic Control Profiles (TCPs).
Interface Set (IFL-Set)	A logical grouping of C-VLANs/IFLs or S-VLANs. Allows aggregate-level shaping and scheduling over a set of IFLs or S-VLANs.
Node	A scheduler node is the entity that manages dequeueing traffic from queues. In H-CoS, there are three levels of scheduling nodes. A node can be a root node, internal node, or a leaf node.
PIR	Peak information rate, also known as a "shaping rate." The PIR/shaping rate specifies maximum bandwidth and is applied to ports, IFL-Sets, and IFLs/VLANs.

Term	Definition
Shaping-rate	See "PIR." The maximum rate a queue, IFL, IFL-Set, or IFD can send at. By default, IFD speed sets the maximum rate.
Scheduler	A class of service CLI container where queue scheduling parameters may be configured.
S-VLAN	Service VLAN, normally the outer VLAN of a dual stacked frame. Used interchangeably with IFL-set as a single S-VLAN often represents a group of C-VLANs. S-VLANs are often associated with aggregation devices such as a DSLAM. See also C-VLAN.
Traffic Control Profile (TCP)	A container for CoS/scheduling parameters designed to provide a consistent way of configuring shaping and guaranteed rates; can be specified at the Port, IFL, and IFL-Set level.
CIR mode	A physical interface is in CIR mode when one of more of its "children" (logical interfaces in this case) have a guaranteed rate configured, but some logical interfaces have a shaping rate configured.
Default mode	A physical interface is in default mode if none of its "children" (logical interfaces in this case) have a guaranteed rate or shaping rate configured.
Excess mode	A physical interface is in excess mode when one of more of its "children" (logical interfaces in this case) has an excess rate configured.
PIR mode	A physical interface is in PIR mode if none of its "children" (logical interfaces in this case) have a guaranteed rate configured, but some logical interfaces have a shaping rate configured.

The H-CoS Reference Model

With all this previous talk of H-CoS, it's time to get down to it. Figure 5-7 provides the current H-CoS reference model. Recall H-CoS (and currently even per-unit scheduling) is only supported on Trio Q and EQ MPCs.

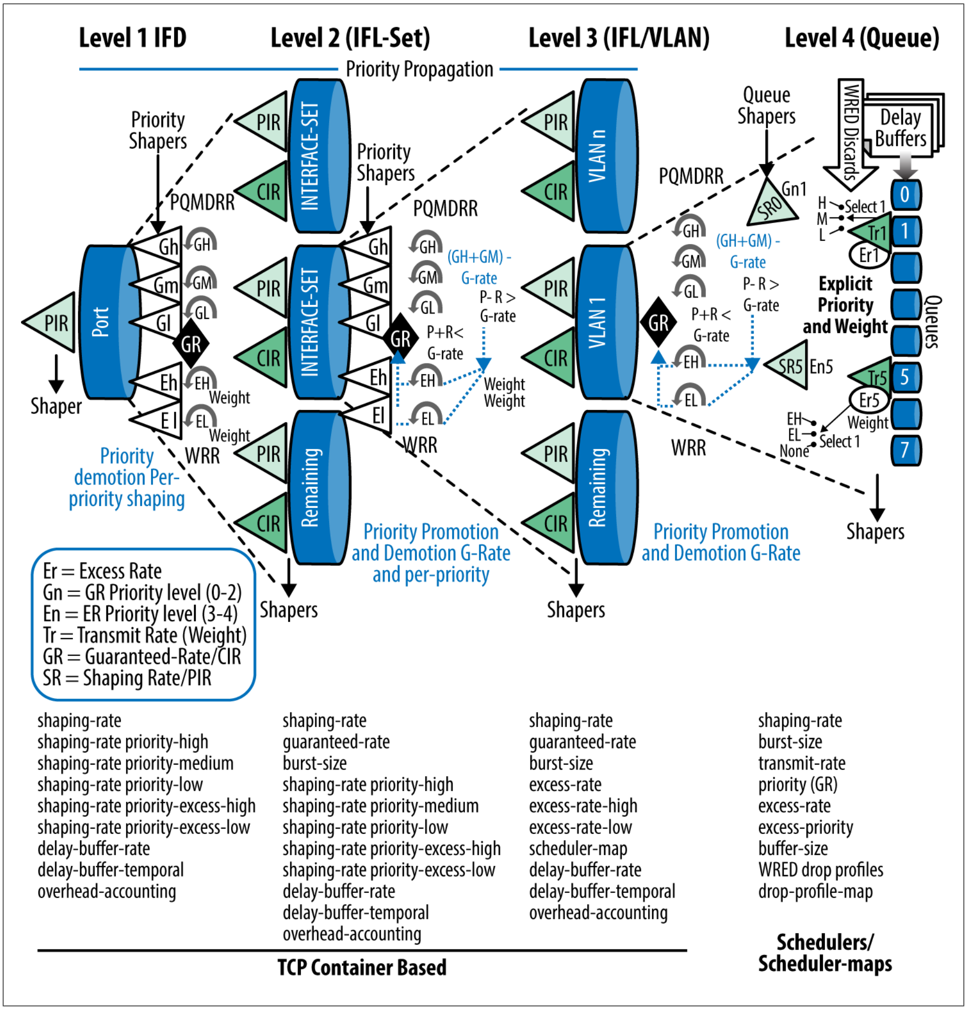


Figure 5-7. The Trio Hierarchical CoS Reference Model.

Let's begin with an overview of some hierarchical scheduling terminology to better facilitate the remainder of the discussion. Scheduler hierarchies are composed of nodes and queues. Queues terminate the scheduler hierarchy and are therefore always at the top. Queues contain the notification cells that represent packets pending egress on some interface. Below the queues, there are one or more scheduler nodes. A scheduler node's position in the hierarchy is used to describe it as a root node, a leaf node, or nonleaf node; the latter is also referred to as an *internal node*. Leaf nodes are the highest scheduling nodes in the hierarchy, which means they are the closest to the queues. As the name implies, an internal node is positioned between two other scheduling nodes. For example, an interface-set, which is shown at level 2 in Figure 5-7, is above a physical port (root) node and below the leaf node at the logical interface level, making it an internal node.

The figure is rather complicated, but it amounts to a virtual Rosetta stone of Junos H-CoS, so it has to be. Each level of the H-CoS hierarchy is described individually to keep things in manageable chunks.

Level 4: Queues

Figure 5-7 represents a four-level scheduling hierarchy in conventional format, which places the IFD on the left for a horizontal view or at the bottom of the figure for vertically oriented diagrams. As the discussion surrounds egress queuing, it can be said that things begin on the right, where the queues are shown at level 4. The egress queues hold the notification cells that are in turn received over the switch fabric. Ingress queue, when supported, worked in the opposite direction, ultimately holding notification cells received over a WAN port and now pending transmission over the switch fabric.

The queue level of the hierarchy is always used in any CoS model, be it port mode, per unit, or hierarchical.

Trio MPCs support eight user queues, each with a buffer used to store incoming notifications cells that represent packets enqueued for transmission over the IFD. Note that WRED actions in Trio occur at the tail of the queue, rather than at the head, which means a packet that is selected for discard is never actually queued.

Queue-level configuration options are shown below the queues at the bottom of Figure 5-7. These parameters allow you to configure each queue's transmit (guaranteed rate), scheduling priority, excess priority, buffer depth, and WRED settings.

shaping rate
: Optional: This parameter places a maximum limit on a queue's transmit capacity. The differences between the transmit rate and shaping rate is used to accommodate excess traffic. By default, shaping rate is equal to the interface speed/shaping rate, which means a queue is allowed to send at the full rate of the interface.

burst size
: Optional: You can manage the impact of bursts of traffic on your network by configuring a burst size value with a queue's shaping rate. The value is the maximum bytes of rate credit that can accrue for an idle queue (or another scheduler node). When a queue or node becomes active, the accrued rate credits enable the queue or node to catch up to the configured rate. The default is 100 milliseconds. Burst size is detailed in a subsequent section.

transmit-rate
: Optional: Defines a queue's transmit weight or percentage. Defines the guaranteed rate for the queue, assuming no priority-based starvation occurs. When no transmit weight is specified, or when the transmit rate is reached, the queue can only send excess-rate traffic as that queue's priority is demoted to the excess region. Note

that strict-high cannot exceed its transmit weight of 100% and therefore is never subject to queue level demotion.

> Options to `transmit-rate` include `exact`, `remainder`, and `rate-limit`. Use `exact` to prevent a queue from exceeding the configured transmit weight, in effect imposing a shaper where PIR is equal to transmit weight. As with any shaper, excess traffic is buffered and smoothed, at the expense of added latency as a function of buffer depth. Given this, you cannot combine the `shaping-rate` statement with `exact`.
>
> The `remainder` option gives the queue a share of any unassigned transmit rate for the interface. If the sum of configured transmit weight is 80%, then a single queue set to remainder will inherit a transmit rate of 20%. The `rate-limit` option uses a policer to prevent a queue from exceeding its transmit rate, trading loss for delay and making its use common for LLQ applications like VoIP.

priority
: Optional: Sets a queues scheduler priority to one of three levels for guaranteed rate traffic. Default is guaranteed-low when not set explicitly.

excess-rate
: Optional: Defines a queue's weight as either a percentage, or a proportion, for any unused bandwidth. Behavior varies based on interface mode, explicit configuration, and whether any other queues have an explicit weight configured. By default, excess bandwidth between the guaranteed and shaped rate is shared equally among queues. If none of the queues have an excess rate configured, then the excess rate will be the same as the transmit rate percentage. If at least one of the queues has an excess rate configured, then the excess rate for the queues that do not have an excess rate configured will be set to zero.

excess-priority
: Optional: Set one of two priorities for excess rate traffic, or none to prevent the queue from sending any excess rate traffic. This behavior results in the queue being forced to buffer any traffic that exceeds the configured G-Rate. By default, excess priority matches normal priority such that H/SH get EH while all other get EL.

buffer-size
: Optional: This parameter allows you to specific an explicit buffer size, either as a percent of interface speed or as a function of time (specified in microseconds); the latter option is popular for real-time or low-latency queues (LLQ). By default, buffer size is set to a percentage that equals the queue's transmit rate.

WRED drop-profiles
: Optional: Drop profiles define WRED values to define one or more queue fill level to drop probability points. While not defined at the queue level, drop profiles are mapped to a specific queue using the **drop-profile-map** statement.

drop-profile-map
: Optional: Drop profile maps tie one or more WRED drop profiles to a queue. The default WRED profile is used when no explicit drop profile mapping is specified.

In Figure 5-7, queues 1 and 5 are detailed to show that both have a transmit and excess rate configured; the focus is on the guaranteed rate at queue 1 and the excess rate at queue 5. At queue 1, guaranteed rate traffic is sent at one of the three normal or guaranteed priorities, based on the queue's configuration. In like fashion, queue 5 is configured to use one of two excess rate priorities along with an excess weight that is used to control each queue's share of remaining bandwidth. In both cases, a queue-level shaper is supported to place an absolute cap on the total amount of guaranteed + excess rate traffic that can be sent.

In the current implementation, WRED congestion control is performed at the queue level only; if a queue's WRED profile accepts a packet for entry to the queue, no other hierarchical layer can override that decision to perform a WRED discard.

H-CoS does not alter the way you assign these parameters to queues. As always, you define one or more schedulers that are then linked to a queue using a scheduler map. However, you will normally link the scheduler map through a TCP rather than applying it directly to the IFL. For example:

```
{master}[edit]
jnpr@R1-RE0# show class-of-service schedulers sched_ef_50
transmit-rate percent 50 rate-limit;
buffer-size temporal 25k;
priority strict-high;
```

In this case, the scheduler is named `sched_ef_50` and defines a transmit rate, priority, and temporal buffer size. The `rate-limit` option is used to prevent this strict-high queue from starving lesser priority queues through a policer rather than buffering (shaping) mechanism. The scheduler does not have a shaping rate, but the combination of transmit rate with rate limit caps this scheduler to 50% of the IFD rate.

Next, the `sched_map_pe-p` scheduler map in turn links this scheduler, along with the six others, to specific forwarding classes (FCs), which in Junos are synonymous with queues.

```
{master}[edit]
jnpr@R1-RE0# show class-of-service scheduler-maps
sched_map_pe-p {
    forwarding-class ef scheduler sched_ef_50;
    forwarding-class af4x scheduler sched_af4x_40;
    forwarding-class af3x scheduler sched_af3x_30;
    forwarding-class af2x scheduler sched_af2x_10;
    forwarding-class af1x scheduler sched_af1x_0;
```

```
        forwarding-class be scheduler sched_be_5;
        forwarding-class nc scheduler sched_nc;
    }
```

Explicit Configuration of Queue Priority and Rates

Before moving down the hierarchy into level 3, it should be stressed that priority and transmit rate/excess weights are explicitly configured for queues via a scheduler definition and related scheduler map. Other areas of the hierarchy use inherited or propagated priority to ensure that all levels of the hierarchy operate at the same priority at any given time, a priority that is determined by the highest priority queue with traffic pending.

Level 3: IFL

Still referring to the figure, it's clear that in a four-level hierarchy, queues feed into logical interfaces or IFLs (Interface Logical Levels). On an Ethernet interface, each VLAN represents a separate IFL (subinterface), so the terms IFLs and VLANs are often used interchangeably, but given the MX also supports WAN technologies this could also be a Frame-Relay DLCI, making IFL the more general term. The figure shows that the first set of schedulers is at level 3 and that there is a discrete scheduler for each of the five scheduling priorities supported in Trio. These schedulers service the queues based on their current priority, starting with high and working down to excess-low, as described in the section titled Trio Scheduling.

Level 3 has its own set of traffic conditioning parameters, including a guaranteed (CIR), shaping rate (PIR), and burst size. In contrast to queues, you use `traffic-control-profiles` (TCPs) to specify traffic parameters at levels 1 to 3. The TCP used at the node below queues, which is the IFL level in this example, specifies the scheduler map that binds specific schedulers to queues.

The IFL level of the hierarchy is used in per-unit and hierarchical CoS. When in port mode, a dummy L3 (and L2) node is used to link the IFD to the queues. In per-unit or hierarchical mode, multiple level 3 IFLs, each with their own set of queues and schedulers, can be mapped into a shared level 2 node.

The Guaranteed Rate

With H-CoS, you can configure guaranteed bandwidth, also known as a committed information rate (CIR), at Layer 2 or Layer 3 nodes. You configure CIR and PIR parameters within a traffic control profile, which you then attach to the desired L2/L3 node to instantiate the selected CIR and PIR. A TCP attached to a L1/IFD node can perform shaping to a peak rate only; guaranteed rates are not supported at L1 of the hierarchy.

The guaranteed rate is the minimum bandwidth the queue should receive; if excess physical interface bandwidth is available for use, the logical interface can receive more than the guaranteed rate provisioned for the interface, depending on how you choose to manage excess bandwidth and the interface's mode of PIR versus CIR/PIR, as explained in the following.

The queue transmit rate is considered a form of guaranteed rate, but assigning a transmit rate to a queue is not the same thing as allocating G-Rate bandwidth to a scheduler node. Assigning a G-Rate is optional. You can use shaping rates alone to define a PIR service, which is similar to a 0 CIR Frame Relay service. In PIR mode, there is no guarantee, only maximum rated limits. Still, a queue expects to get its transmit rate before it is switched to an excess priority level. The CIR mode extends the PIR model by allowing you to allocate reserved bandwidth that ideality is dimensions based on aggregate queue transmit rates.

Junos does allow overbooking of CIRs in H-CoS mode. However, just because you can does not mean you should. To ensure that all nodes and queues can achieve their G-Rates, you should ensure that G-Rates are not overbooked. Overbooking is not supported in per unit mode and is not applicable to port mode CoS.

Clearly, you cannot expect to get simultaneous G-Rate throughput among all queues in an overbooked CIR configuration; still, customers have asked for the capability. Some external mechanism of controlling sessions/traffic flow known as Connection Admission Control (CAC) is needed in these environments if guarantees are to be made, in which case the CAC function ensures that not all queues are active at the same time, such as might be the case in a Video on Demand (VoD) service. Overbooking is not supported in per unit scheduling mode, making it necessary that the sum of IFL CIRs never exceed the IFD shaping rate.

In per-unit mode, you cannot provision the sum of the guaranteed rates to be more than the physical interface bandwidth. If the sum of the guaranteed rates exceeds the interface shaping bandwidth, the commit operation does not fail, but one or more IFLs will be assigned a G-Rate of 0, which can dramatically affect their CoS performance if they are competing with other IFLs on the same IFD that were able to receive their G-Rate. H-CoS mode permits G-Rate overbooking without any automatic scaling function.

When using per unit scheduler and CIR mode, be sure to factor the G-Rate bandwidth that is allocated to the control scheduler. This entity is automatically instantiated on VLAN tagged interface using unit 32767 and is used to ensure that LACP control traffic can still be sent to a remote link partner, even if the egress queues are in a congested state. In the v11.4R1 release, this scheduler was provisioned with 2 Mbps of G-Rate bandwidth, so you may need to increase IFD shaping rate by that amount when the sum of user allocated IFL G-Rates are within 2Mbps of the IFD shaping rate. Details on the control scheduler are provided later in the CoS Lab section.

Priority Demotion and Promotion

Scheduler and queues handle priority promotion and demotion differently. Scheduler nodes at levels 2 and 3 perform *both* priority promotion and priority demotion, a function that is based on two variables. The primary demotion behavior relates to the node's configured guaranteed rate versus the aggregate arrival rate of G-Rate traffic at that node; G-Rate traffic is the sum of GH, G, and GM priority levels. Priority promotion does not occur at level 1, the IFD level, given that node supports peak-rate shaping only and does not understand the notion of a guaranteed rate.

A second form of demotion is possible based on per priority shaping, occurring when a given traffic priority exceeds the node's per priority shaper. Per priority shaping demotion applies to GH, GM, and GL traffic, but you can block node-level priority shaping-based demotion of GH/GM by setting the queue to an excess priority of none.

G-Rate Based Priority Handling at Nodes. As shown in Figure 5-7, all GH and GM traffic dequeued at a given level 2/3 node is subtracted from the node's configured guaranteed rate. The remaining guaranteed bandwidth is used to service GL traffic; if GL and GM traffic alone should exceed the node's G-Rate, the node begins accruing negative credit; G-Rate demotion does not affect GH and GM, which means queues at this priority should always get their configured transmit rate, which will be based on the IFL's G-Rate. When the G-Rate is underbooked, credits will remain to allow the node to handle GL traffic, and it is possible to even promote some excess traffic into the guaranteed region, as described in the following.

In the event that the node's GL load exceeds the remaining G-Rate capacity (an event that can occur due to priority promotion, as described next, or because of simple G-Rate overbooking), the excess GL traffic is demoted to either EH or EL, based on the queue's `excess-priority` setting as a function of priority inheritance. Queues that have no explicit excess priority setting default to a mapping that has strict high and high going into excess-high while medium and low map to the excess low region.

> Because a scheduler node *must* be able to demote EL to ensure that the guaranteed path is not blocked in the event of overbooked G-Rates, you cannot combine the `excess-priority none` statement with a queue set to low priority.

In reverse fashion, excess traffic can be promoted into the guaranteed region when L2 or L3 nodes have remaining G-Rate capacity after servicing all G-Rate queues. In the case of promotion, both excess high and low are eligible, but even in their new lofty stations, the node's scheduler always services real GL before any of the promoted excess traffic, a behavior that holds true in the case of GL demotion as well; the effect is that the relative priority is maintained even in the face of promotion or demotion.

Per Priority Shaping–Based Demotion at Nodes. H-CoS supports shaping traffic at L1, L2, and L3 scheduler nodes based on one of five priority levels (three used for the guaranteed region and two for operating in the excess region), thus you can configure up to five per priority shapers at each node. Traffic in excess of the per priority shaper is demoted, again based on the queue's excess priority setting. When a queue is set to an excess priority of none, it prevents demotion at a per priority shaper, which forces the queue to stop sending and begin buffering in that case.

Queue-Level Priority Demotion. It's important to note that a queue demotes its own GH, GM, or GL as a function of exceeding the configured transmit rate. In effect, a queue's transmit rate is its G-Rate, and it handles its own demotion and promotion based on whether it's sending at or below that rate. The queue demotes traffic in excess of its transmit rate to the configured excess level, and that demoted priority is in turn inherited by scheduler nodes at lower levels, a function that is independent of scheduler node promotion and demotion, as described previously.

A queue promotes its traffic back to the configured G-Rate priority whenever its again transmitting at, or below, its configured rate. A queue that is blocked from using excess levels appears to simply stop sending when it reaches its configured transmit rate.

Because the priority inheritance scheme is used to facilitate priority demotion setting, a queue to `excess-rate none` prevents demotion at subsequent scheduling levels. Such a queue is forced to buffer traffic (or discard if rate limited) rather than being demoted to an excess region.

Based on configuration variables, it's possible that such a queue may be starved; this is generally considered a feature, and therefore proof that things are working as designed. The software does not guarantee to catch all cases where such a queue may be starved, but a commit fail is expected if you configure excess none for a queue that is also configured with a transmit-rate expressed as a percent, when the parent's guaranteed rate is set to zero (i.e., the IFD is in PIR mode). This is because such a queue has no guaranteed rate and can only send at the excess level and so would be in perpetual starvation. PR 778600 was raised during testing when the expected commit error for this configuration was not generated.

Level 2: IFL-Sets

Still referring to Figure 5-7, the next stage in the hierarchy is level 2, the mystical place where IFL-Sets are to be found. All that was described previously for level 3 holds true here also, except now we are scheduling and shaping an aggregate set of IFLs or a list of outer VLANs; the degenerate case is a set with a single IFL or VLAN, which yields a set of one. An IFL-Set is an aggregation point where you can apply group-level policies to control CIR, PIR, and how each set shares any excess bandwidth with other nodes at the same hierarchy level (other IFL-Sets). In the PIR mode, the sum of queue transmit rates should be less than or equal to the L2 nodes shaping rate. By increasing the L2 nodes shaping rates, you make more excess bandwidth available to the IFLs that attach to it. In CIR mode, the sum of queue transmit rates should be less than or equal to the node's guaranteed rate. By assigning a shaping rate that is higher, you are again providing excess bandwidth to the IFLs and their queues.

Remaining Traffic Profiles

One of the IFLs in the figure is associated with a special traffic profile called *remaining*; the same construct can also be found for the IFD at level 1. The ability to define a set of traffic scheduling and shaping/CIR parameters for IFLs that otherwise have no explicit scheduler setting of their own is a powerful Trio feature that is detailed in a later section. For now, it's sufficient to say that IFLs can be automatically mapped into a shared level 2 scheduler associated with a specific IFL-Set to catch IFLs that are listed as a member of that set yet which have no scheduler settings applied. And, one so grouped all such IFLs are then treated to a shared set of queues and a common TCP profile.

> The IQ2E card does not support the notion of a remaining CoS profile, forcing you to explicitly configure all IFLs for some level of CoS.

The remaining traffic profile construct is supported at the IFD level as well; as with level 2, it's again used to provide a default CoS-enabled container for IFLs, but this profile catches those IFLs that are not placed into any IFL-Set and which also do not have their own scheduler settings applied (either directly or through a scheduler map within a TCP container).

The IFL-Set level also supports priority-based shaping, which allows you to shape at each of the five priority levels. As shown, priority-level shaping is in addition to the aggregate shaping for all traffic at the node.

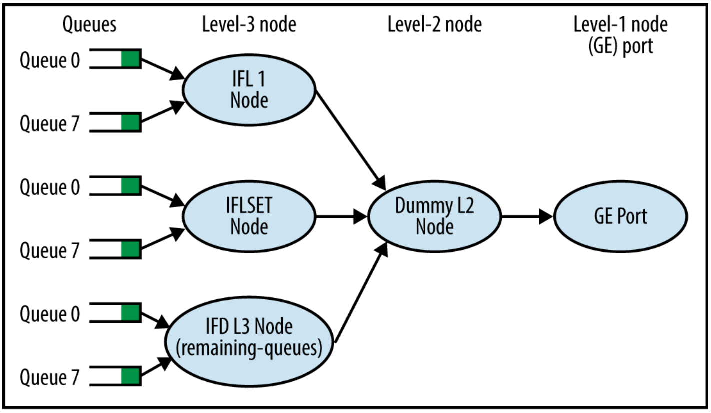
Forcing a Two-Level Scheduling Hierarchy

In some cases, users may opt for a reduction in the number of scheduler hierarchies to promote better scaling. When operating in per unit scheduling mode, all logical interfaces share a common dummy level 2 scheduling node (one per port). In contrast, when in the full-blown hierarchical scheduling mode, each logical interface can have its own level 2 node, which means that a key scaling factor for H-CoS is the total number of level 2 nodes that can be allocated.

When in hierarchical scheduling mode, you can limit the number of scheduling levels in the hierarchy to better control system resources. In this case, all logical interfaces and interface sets with a CoS scheduling policy share a single (dummy) level 2 node, so the maximum number of logical interfaces with CoS scheduling policies is increased to the scale supported at level 3, but this forces IFL-sets to be at level 3, the cost being that in two-level mode you lose the ability to have IFLs over IFL-Sets and cannot support IFL-Set-level remaining queues.

> The system tries to conserve level 2 scheduler nodes by default; a level 2 scheduler node is only created for IFL-Sets when any member IFL has `traffic-control-profile` configured, or the `internal node` command is used at the `[edit class-of-service]`hierarchy:
>
> ```
> class-of-service interfaces {
>         interface-set foo internal-node
> }
> ```

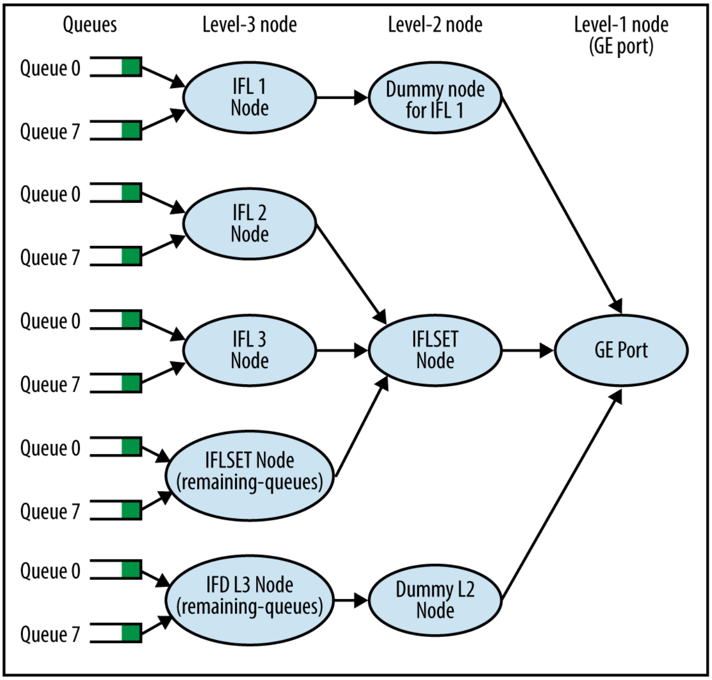
Figure 5-8 illustrates the impacts of a dummy L2 scheduling node that results from a two-level hierarchy.



- Supports IFLSET/IFL based Queuing:
- Port level remaining-queues
- 8 Queues per IFLSET (group of IFLs)
- IFLS over IFLSET NOT supported
- 8 Queues per IFL
- Scheduling nodes per Port:
    - 1 L1 node
    - 1 L2 node
    - 1 L3 node / 8 Queues
- Scheduling nodes per IFLSET:
    - 1 L3 node / 8 Queues
- Scheduling nodes per IFL (IFD as parent):
    - 1 L3 node / 8 Queues

Figure 5-8. Two-Level Scheduling Hierarchy.

Note that in two-level mode, only IFD/port-level remaining queues are supported, and that all IFL-Sets must be defined at level 3. Figure 5-9 provides a similar view of full-blown H-CoS, with its three-level scheduling hierarchy, to provide clear contrast to the previous two-level scheduling hierarchy.



- Supports IFLSET/IFL based Queuing:
- Port level remaining-queues
- IFLSET (group of IFLs) support with remaining-queues
- 8 Queues per IFL
- Scheduling nodes per Port:
    - 1 L1 node
    - 1 L2 node
    - 1 L3 node / 8 Queues

- Scheduling nodes per IFLSET:
    - 1 L2 node
    - 1 L3 node / 8 Queues
- Scheudling nodes per IFL (IFLSET as parent):
    - 1 L3 node / 8 Queues
- Scheduling nodes per IFL (IFD as parent):
    - 1 L2 node
    - 1 L3 node / 8 Queues

Figure 5-9. A Three-Level Scheduling Hierarchy.

To configure scheduler node scaling, you include the `hierarchical-schedulers` statement with the `maximum-hierarchy-levels` option at the `[edit interfaces xe-fpc/pic/port]` hierarchy level. In the v11.4R1 release, the only supported value is 2.

```
{master}[edit interfaces xe-2/0/0]
jnpr@R1-RE0# set hierarchical-scheduler maximum-hierarchy-levels ?
Possible completions:
<maximum-hierarchy-levels>  Maximum hierarchy levels (2..2)
{master}[edit interfaces xe-2/0/0]
jnpr@R1-RE0# set hierarchical-scheduler maximum-hierarchy-levels 2

{master}[edit interfaces xe-2/0/0]
jnpr@R1-RE0# show
hierarchical-scheduler maximum-hierarchy-levels 2;
vlan-tagging;
. . .
```

Level 1: IFD

Again referring to Figure 5-7, the next stage in the hierarchy is level 1, the physical interface or IFD level. The IFD level is a bit unique when compared to levels 2 and 3 in that it does not support a CIR (G-Rate), which in turn means that G-Rate-based priority promotion and demotion does not occur at the IFD level. PIR shaping is supported, both as an aggregate IFD rate in addition to five levels of per priority shaping.

Remaining

The user may have some traffic that is not captured by explicit class of service configuration at various levels of the hierarchy. For example, the user may configure three logical interfaces over a given S-VLAN set (level 2), but apply a traffic control profile to only one of the C-VLANs/IFLs at level 3. Traffic from the remaining two C-VLANs/IFLs is considered "unclassified traffic." In order for the remaining traffic to get transmit rate guarantees, the operator must configure an `output-traffic-control-profile-remaining` to specify a guaranteed- and shaping-rate for the remaining traffic. In the absence of this construct, the remaining traffic gets a default guaranteed rate of 0 bps, or not much guarantee at all. You can limit, or cap, the total remaining traffic by including the `shaping-rate` statement. As with any TCP, you can also alter the `delay-buffer-rate` to control the size of the delay buffer for remaining traffic, when desired.

Junos H-CoS supports remaining TCPs at the IFL-Set and IFD hierarchy levels; the former captures remaining VLANs for a given IFL-Set, whereas the latter is used to capture all leftover VLANs that are not part of any IFL-Set.

> If you don't configure a remaining scheduler, unclassified traffic is given a minimum bandwidth that is equal to two MTU-sized packets.

Remaining Example

Figure 5-10 shows how remaining scheduler nodes are used to provide CoS for otherwise unconfigured IFLs/C-VLANs. In this example, the C-VLANs are captured as part of an IFL-Set called `iflset_1`.

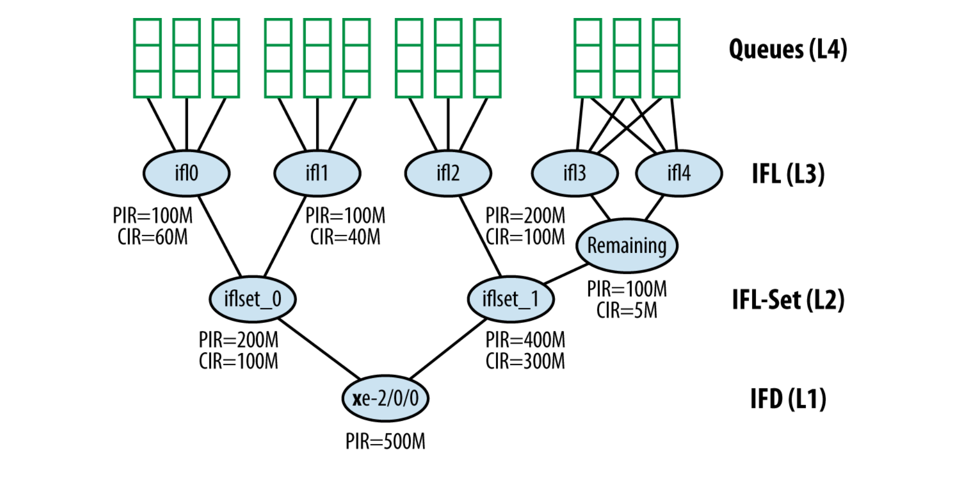


Figure 5-10. Remaining C-VLANs in an IFL-Set.

To make this happen, the user has configured logical interfaces 0 to 4 under an `interface-set` at the `[edit interfaces interface-set]` hierarchy. However, a traffic control profile is only attached to a subset of the IFLs at the`[edit class-of-service interfaces]` hierarchy, namely IFLs 0 and 2. IFLs 3 and 4, which don't have traffic control profile attached, are called "remaining" traffic. In this case, the remaining scheduler is used to capture these IFLs, with the result being they share a set of queues and the CoS profile of the associated level 3 scheduler node. Here, two IFL-Sets are defined to encompass the IFD's 5 logical units:

```
{master}[edit]
jnpr@R1-RE0# show interfaces interface-set iflset_0
interface xe-2/0/0 {
    unit 0;
    unit 1;
}

{master}[edit]
jnpr@R1-RE0# show interfaces interface-set iflset_1
interface xe-2/0/0 {
    unit 2;
    unit 3;
    unit 4;
}
```

While interface set 1 has three IFLs, a TCP is applied to only one of them; note that both of the IFLs in IFL-Set 0 have TCPs applied:

```
{master}[edit]
jnpr@R1-RE0# show class-of-service interfaces xe-2/0/0
output-traffic-control-profile 500m_shaping_rate;
unit 0 {
    output-traffic-control-profile tc-ifl0;
}
unit 1 {
    output-traffic-control-profile tc-ifl1;
}
unit 2 {
    output-traffic-control-profile tc-ifl2;
}
```

Meanwhile, the two interface sets are linked to TCPs that control the level 2 scheduler node's behavior:

```
jnpr@R1-RE0# show class-of-service interfaces interface-set iflset_0
output-traffic-control-profile tc-iflset_0;

{master}[edit]
jnpr@R1-RE0# show class-of-service interfaces interface-set iflset_1
output-traffic-control-profile tc-iflset_1;
output-traffic-control-profile-remaining tc-iflset_1-remaining;
```

The key point is that IFL-Set 1 uses the `output-traffic-control-profile-remaining` keyword to link to a second TCP that is used to service any IFLs in the named set that do not have explicit TCP configuration, thus matching the example shown in Figure 5-10. Note that this remaining profile links to a scheduler map that is used to provide schedulers for the shared set of queues that are shared by all remaining IFLs in this set:

```
{master}[edit]
jnpr@R1-RE0# show class-of-service traffic-control-profiles tc-iflset_1-remaining
scheduler-map smap-remainder;
shaping-rate 100m;
```

Be sure to include a scheduler map in your remaining traffic profiles to ensure things work properly. Without a map, you may not get the default scheduler and so end up without any queues.

This example uses a customized scheduler for remaining traffic that has only two FCs defined. If desired, the same scheduler map as used for the IFLs could be referenced. The scheduler map is displayed for comparison:

```
{master}[edit class-of-service scheduler-maps]
jnpr@R1-RE0# show
sched_map_pe-p {
    forwarding-class ef scheduler sched_ef_50;
    forwarding-class af4x scheduler sched_af4x_40;
    forwarding-class af3x scheduler sched_af3x_30;
    forwarding-class af2x scheduler sched_af2x_10;
    forwarding-class af1x scheduler sched_af1x_5;
    forwarding-class be scheduler sched_be_5;
    forwarding-class nc scheduler sched_nc;
    forwarding-class null scheduler sched_null;
}
smap-remainder {
    forwarding-class be scheduler sched_be_5;
```

```
        forwarding-class nc scheduler sched_nc;
    }
    . . .
```

Figure 5-11 goes on to demonstrate how remaining is used at the IFD level to capture VLANs/IFLs that are not part of any IFL-Set.

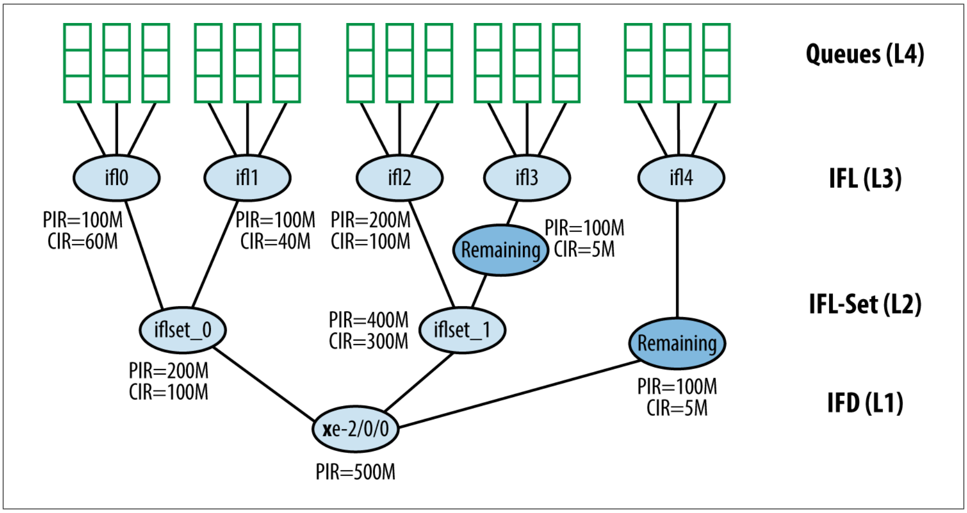


Figure 5-11. Remaining C-VLANs in an IFD.

It's much the same principle as the previous IFL-Set remaining example, but in this case you apply the remaining TCP, here called `tc-xe-2/0/0_remaining`, to the IFD itself:

```
    jnpr@R1-RE0# show class-of-service interfaces xe-2/0/0
    output-traffic-control-profile 500m_shaping_rate;
    output-traffic-control-profile-remaining tc-xe-2/0/0_remaining;
    unit 0 {
        output-traffic-control-profile tc-ifl0;
    }
    unit 1 {
        output-traffic-control-profile tc-ifl1;
    }
    unit 2 {
        output-traffic-control-profile tc-ifl2;
    }
```

The new TCP, which is used by IFLs/VLANs that *are not* assigned to any sets, is shown:

```
    {master}[edit]
    jnpr@R1-RE0# show class-of-service traffic-control-profiles tc-xe-2/0/0_remaining
    scheduler-map smap-remainder;
    shaping-rate 100m;
    guaranteed-rate 5m;
```

Note again the inclusions of a `scheduler-map` statement in the remaining TCP; the map is used to bind schedulers to the shared set of queues.

To put the IFD-level set remaining TCP to use, the previous configuration is modified to remove IFL 4 from iflset_1:

```
{master}[edit]
jnpr@R1-RE0# show interfaces interface-set iflset_1
interface xe-2/0/0 {
    unit 2;
    unit 3;
}
```

Note this interface inherited its scheduler map from the TCP applied to the IFL-Set, and thereby had eight queues, EF set to SH, etc. Just as with an IFL-Set remaining profile, you can include a scheduler map statement in the remaining profile, unless you want the default scheduler for this traffic.

The configuration results in the scheduling hierarchy shown in Figure 5-11. To summarize, we now have two interface sets and five IFLs, but IFL 4 does not belong to either set. The two IFLs in IFL-Set 0 both have a TCP applied and so both have their own level 3 scheduler node. IFL 3, in contrast, is assigned to IFL-Set 1 but does not have a TCP, and therefore uses the remaining TCP for that IFL-Set, here called `tc-iflset_1-remaining`. IFL 4 does not belong to any IFL-Set, nor does it have a TCP attached. As such it's caught by the remaining traffic profile at the L1 node. The resulting scheduler hierarchy is confirmed:

```
NPC2(R1-RE0 vty)# show cos scheduler-hierarchy

class-of-service EGRESS scheduler hierarchy - rates in kbps
---------------------------------------------------------------------------------
                                    shaping guarntd delaybf  excess
interface name               index    rate    rate    rate    rate      other
---------------------------- -----  ------- ------- ------- ------- -------------
xe-2/0/0                       148   500000       0       0       0
  iflset_0                      85   200000  100000       0       0
    xe-2/0/0.0                 332   100000   60000       0       0
      q 0 - pri 0/0          44600        0     10%       0      0%
      q 1 - pri 0/0          44600        0     10%       0      0%
      q 2 - pri 0/0          44600        0     10%       0      0%
      q 3 - pri 3/0          44600        0     10%       0      0%
      q 4 - pri 0/0          44600        0     10%       0      0%
      q 5 - pri 4/0          44600        0     40%   25000      0% exact
      q 6 - pri 0/0          44600        0     10%       0      0%
      q 7 - pri 2/5          44600        0       0       0      0%
    xe-2/0/0.1                 333   100000   40000       0       0
      q 0 - pri 0/0          44600        0     10%       0      0%
      q 1 - pri 0/0          44600        0     10%       0      0%
      q 2 - pri 0/0          44600        0     10%       0      0%
      q 3 - pri 3/0          44600        0     10%       0      0%
      q 4 - pri 0/0          44600        0     10%       0      0%
      q 5 - pri 4/0          44600        0     40%   25000      0% exact
      q 6 - pri 0/0          44600        0     10%       0      0%
      q 7 - pri 2/5          44600        0       0       0      0%
    iflset_0-rtp                85   500000       0       0       0
```

```
      q 0 - pri 0/1              2       0     95%     95%      0%
      q 3 - pri 0/1              2       0      5%      5%      0%
  iflset_1                      86  400000  300000       0       0
    xe-2/0/0.2                 334  200000  100000       0       0
      q 0 - pri 0/0          44600       0     10%       0      0%
      q 1 - pri 0/0          44600       0     10%       0      0%
      q 2 - pri 0/0          44600       0     10%       0      0%
      q 3 - pri 3/0          44600       0     10%       0      0%
      q 4 - pri 0/0          44600       0     10%       0      0%
      q 5 - pri 4/0          44600       0     40%   25000      0% exact
      q 6 - pri 0/0          44600       0     10%       0      0%
      q 7 - pri 2/5          44600       0       0       0      0%
    iflset_1-rtp                86  100000    5000    5000       0
      q 0 - pri 0/0          10466       0     10%       0      0%
      q 3 - pri 3/0          10466       0     10%       0      0%
  xe-2/0/0.32767               339       0    2000    2000       0
    q 0 - pri 0/1                2       0     95%     95%      0%
    q 3 - pri 0/1                2       0      5%      5%      0%
  xe-2/0/0-rtp                 148  100000    5000    5000       0
    q 0 - pri 0/0            10466       0     10%       0      0%
    q 3 - pri 3/0            10466       0     10%       0      0%
```

Of note here is how both the IFL-Set and IFD remaining profiles (`rtp`) are shown with the two queues that result from the user-specified scheduler map. In testing, it was found that without the scheduler map statement in the remaining profile, the RTPs were displayed as if no remaining profile was in effect at all. Compare the below to the output shown above, when the scheduler maps are applied to remaining profiles, and note the absence of queues next to the RTPs:

```
{master}[edit]
jnpr@R1-RE0# show | compare
[edit class-of-service traffic-control-profiles tc-iflset_1-remaining]
-    scheduler-map smap-remainder;
[edit class-of-service traffic-control-profiles tc-xe-2/0/0_remaining]
-
NPC2(R1-RE0 vty)# sho cos scheduler-hierarchy

class-of-service EGRESS scheduler hierarchy - rates in kbps
--------------------------------------------------------------------------------
                                   shaping guarntd delaybf  excess
interface name               index    rate    rate    rate    rate      other
---------------------------- ----- ------- ------- ------- ------- -------------
xe-2/0/0                       148  500000       0       0       0
  iflset_0                      85  200000  100000       0       0
    xe-2/0/0.0                 332  100000   60000       0       0
      q 0 - pri 0/0          44600       0     10%       0      0%
      q 1 - pri 0/0          44600       0     10%       0      0%
      q 2 - pri 0/0          44600       0     10%       0      0%
      q 3 - pri 3/0          44600       0     10%       0      0%
      q 4 - pri 0/0          44600       0     10%       0      0%
      q 5 - pri 4/0          44600       0     40%   25000      0% exact
      q 6 - pri 0/0          44600       0     10%       0      0%
      q 7 - pri 2/5          44600       0       0       0      0%
    xe-2/0/0.1                 333  100000   40000       0       0
      q 0 - pri 0/0          44600       0     10%       0      0%
      q 1 - pri 0/0          44600        0     10%       0      0%
      q 2 - pri 0/0          44600        0     10%       0      0%
      q 3 - pri 3/0          44600        0     10%       0      0%
      q 4 - pri 0/0          44600        0     10%       0      0%
      q 5 - pri 4/0          44600        0     40%   25000      0% exact
      q 6 - pri 0/0          44600        0     10%       0      0%
      q 7 - pri 2/5          44600        0       0       0      0%
    iflset_0-rtp                85   500000       0       0       0
  iflset_1                      86   400000  300000       0       0
    xe-2/0/0.2                 334   200000  100000       0       0
      q 0 - pri 0/0          44600        0     10%       0      0%
      q 1 - pri 0/0          44600        0     10%       0      0%
      q 2 - pri 0/0          44600        0     10%       0      0%
      q 3 - pri 3/0          44600        0     10%       0      0%
      q 4 - pri 0/0          44600        0     10%       0      0%
      q 5 - pri 4/0          44600        0     40%   25000      0% exact
      q 6 - pri 0/0          44600        0     10%       0      0%
      q 7 - pri 2/5          44600        0       0       0      0%
    iflset_1-rtp                86   100000    5000    5000       0
  xe-2/0/0.32767               339        0    2000    2000       0
    q 0 - pri 0/1                2        0     95%     95%      0%
    q 3 - pri 0/1                2        0      5%      5%      0%
  xe-2/0/0-rtp                 148   100000    5000    5000       0
```

Be sure to reference a scheduler map in remaining profiles to avoid this issue.

Interface Modes and Excess Bandwidth Sharing

Interfaces are said to operate in either a Committed Information Rate/Peak Information Rate (CIR/PIR) mode, or in PIR-only mode alone. The interface's mode is an IFD-level attribute, which means it has global effects for all IFLs/VLANs and scheduler nodes configured on the interface. The IFD mode is determined by the guaranteed rate configuration (or lack thereof) across all children/grandchildren on the IFD. If all descendants of the IFD are configured with traffic control profiles that specify only a shaping rate with no guaranteed rate, then the interface is said to be operating in the PIR mode.

If any of the descendants are configured with a traffic control profile that has a guaranteed rate, then the interface is said to be operating in the CIR/PIR mode. Switching an interface from PIR to CIR/PIR mode can affect bandwidth distribution among queues. This is because once a G-Rate is set, the concepts of priority promotion and demotion at scheduler nodes come into play. A CIR interface will attempt to meet all GH/GM queue transmit rates, even if they exceed the available G-Rate. The difference is made up by demoting GL queues, even though they have not yet reached their transmit rates, and the balance needed for the GH/GM CIRs comes out of the PIR region, thereby reducing the amount of excess bandwidth available for GL queues. This behavior is demonstrated in a later section.

While a queue's transmit rate is said to be a committed rate/G-Rate, it's not technically the same as a TCP's `guaranteed-rate` statement that is applied to a scheduler node. The former does not place the IFD into CIR mode, whereas the latter does. Still, you could view the case of all queues having a 0 transmit rate, but being allowed to enter excess region, as an extreme case of the PIR mode. In such an extreme case, a queue's excess bandwidth weighting dominates its share of the bandwidth. In contrast, when in CIR mode GH/GM queues with high transmit weights tend to be favored.

PIR Characteristics. As noted previously, an interface is in PIR mode when only shaping rates are configured at the IFL/IFL-Set levels of the hierarchy (levels 2 and 3). The default behavior for this interface mode is a function of scheduler mode and whether or not the interface is overbooked, a condition that occurs when the sum of configured shaping rates exceeds the IFD shaping speed. For PIR mode, the shaping and guaranteed rates are computed as:

```
shaping-rate = 'shaping-rate' if configured or IFD 'port-speed'
guaranteed-rate =
    per-unit mode:
          IFD is under-subscribed?
               shaping-rate is configured?
                     guaranteed-rate = shaping rate, else guaranteed-rate = 1/nth of
                       remaining bw
          IFD is oversubscribed?
               over-subscribed guaranteed-rate = 0
          IFD is in H-Cos?
               hierarchical mode guaranteed-rate = 0
```

PIR/CIR Characteristics. An IFD is in PIR/CIR mode when at least one scheduling node in the hierarchy has guaranteed rate (CIR) configured via a TCP. For this mode, shaping and guaranteed rates are calculated as:

```
shaping-rate = 'shaping-rate' if configured or ifd 'port-speed'
guaranteed-rate = 'guaranteed-rate' if configured, or 0
```

In CIR mode, excess rates are programmed in proportion of guaranteed rates, or in the case of queues, their transmit rates. Scheduler nodes without a guaranteed rate inherit a G-Rate of 0, which affects not only normal transmission, but also excess bandwidth sharing given that the excess bandwidth is shared among nodes at the same hierarchy in proportion to their G-Rates.

Shaper Burst Sizes

MX routers with Trio interfaces support shaping to a peak rate at all levels of the hierarchy through the `shaping-rate` keyword. The shaping rate statement is used a scheduler definition for application to a queue, or in a TCP for shaping at other hierarchies. Levels 2 and 3 of the hierarchy support shaping to a guaranteed (CIR) rate in addition to a peak rate, while queue levels shaping through the `shaping-rate` statement is only to the peak rate.

Shapers are used to smooth traffic to either the guaranteed or peak information rates, in effect trading latency for a smoother output that eases buffering loads on downstream equipment. You can manage the impact of bursts of traffic on your network by configuring a `burst-size` value with either a shaping rate or a guaranteed rate. The value is the maximum bytes of rate credit that can accrue for an idle queue or scheduler node. When a queue or node becomes active, the accrued rate credits enable the queue or node to catch up to the configured rate.

By default, a shaper allows for small periods of burst over the shaping rate to make up for time spent waiting for scheduler service. The default shaping burst size on Trio is based around support for 100 milliseconds of bursting, a value that has been found adequate for most networking environments, where the brief burst above the shaping rate generally does not cause any issues. Figure 5-12 shows the effects of burst size on the shaped traffic rate.

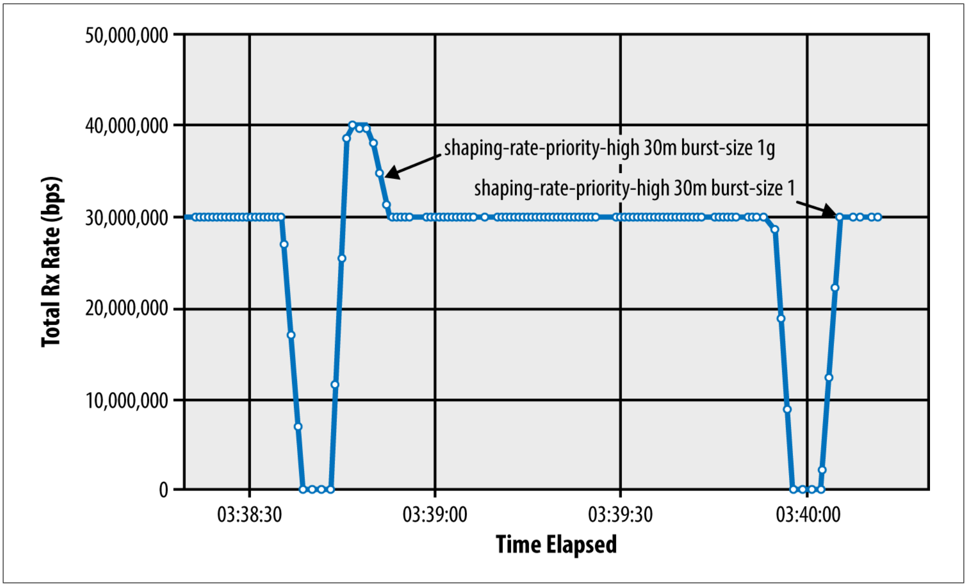


Figure 5-12. Shaper Burst Size Effects.

Figure 5-12 shows two shapers, both set for 30 Mbps. In the first case, a large burst size of 1 GB is set versus the second shaper, which has a very minimal burst size of only 1 byte. In both cases, user traffic has dropped off to 0, perhaps due to lack of activity. Even with constant activity, there are still small delays in a given queue as it waits for a scheduler's attention. The goal of the burst size parameter is to allow a queue to make up for the period of below shaped rate traffic by allowing extra transmit capacity as a function of burst size. As noted, a larger burst value means the queue is able to make up for lost time, as shown by the spike to 40 Mbps, which is well above the shaped rate; while good for the queue or scheduler node in question, a downstream device that has minimal buffers and therefore bases its operation on a smooth shaping rate may well drop some of the excess burst. The low burst rate example on the right shows how smaller burst result in a smoother rate and therefore less buffering needs in downstream equipment.

You can alter the default burst size using the `burst-size` argument to the `shaping-rate` or `guaranteed-rate` statements. You use this statement to specify the desired burst limit in bytes. The supported range for burst size is 0 through 1,000,000,000 bytes (1 GB). While very low burst values can be set, the system always computes a platform-dependant minimum burst size and uses the larger of the two, as described in the following.

> Use caution when selecting a different burst size for your network. A burst size that is too high can overwhelm downstream networking equipment, causing dropped packets and inefficient network operation. Similarly, a burst size that is too low can prevent the network from achieving your configured rate.

If you choose to alter the default burst size, keep the following considerations in mind:

- The system uses an algorithm to determine the actual burst size that is implemented for a node or queue. For example, to reach a shaping rate of 8 Mbps, you must allocate 1 Mbps of rate credits every second. In light of this, a shaping rate of 8 Mbps with a burst size of 500,000 bytes of rate-credit per second is illegal as it only enables the system to transmit at most 500,000 bytes, or 4 Mbps. In general, the system will not install a burst size that prevents the rate from being achieved.
- The minimum and maximum burst sizes can vary by platform, and different nodes and queue types have different scaling factors. For example, the system ensures the burst cannot be set lower than 1 Mbps for a shaping rate of 8 Mbps. To smoothly shape traffic, rate credits are sent much faster than once per second, but the actual interval at which rate credits are sent varies depending on the platform, the type of rate, and the scheduler's level in the hierarchy.
- The system installs the computed minimum burst size if it's larger than the configured burst size. Very small burst sizes are rounded up to the system minimum.
- The guaranteed rate and shaping rate for a given scheduler share the same burst size. If the guaranteed rate has a burst size specified, that burst size is used for the shaping rate; if the shaping rate has a burst size specified, that bursts size is used for the guaranteed rate. If you have specified a burst size for both rates, the system uses the lesser of the two values.
- The system generates a commit error when the burst size configured for the guaranteed rate exceeds the burst size configured for the shaping rate.

- You can configure independent burst size values for each rate, but the system uses the maximum burst size value configured in each rate family. For example, the system uses the highest configured value for the guaranteed rates (GH and GM) or the highest value of the excess rates (EH and EM). The system assigns a single burst size to each of the following rate pairs at each scheduling node:
  - —Shaping and guaranteed rate
  - —Guaranteed high (GH) and guaranteed medium (GM)
  - —Excess high (EH) and excess low (EL)
  - —Guaranteed low (GL)

To provide a concrete example of platform variance for minimum burst size, consider that a Trio Q-type MPC currently supports a minimum burst of 1.837 milliseconds at a L1 and L2 scheduler node for PIR/CIR shaping. On this same card, the minimum burst size for GH/GM priority grouping is 3.674 milliseconds.

In contrast, an MPC EQ-style MPC requires 2.937 milliseconds at the L1 and L2 scheduler levels for PIR/CIR shaping. The increased minimum time/burst rate stems from the fact that an EQ MPC has more scheduling nodes to service, when compared to the Q-style MPC, and therefore it can take longer between scheduling visits for a given node. The longer delay translates to a need to support a larger value for minimum burst size on that type of hardware.

Calculating the Default Burst Size. The default burst size is computed by determining how much traffic can be sent by the highest shaping rate in a pairing (the rate pairs were described previously) in a 100 millisecond period, and then rounding the result up to an exponent of a power of two. For example, the system uses the following calculation to determine the burst size for a scheduler node with a shaping rate of 150 Mbps when an explicit burst size has not been configured:

Max (Shaping rate, guaranteed rate) bps * 100 ms / (8 bits/byte * 1000 ms/s) = 1,875,000 bytes.

The value is then rounded *up* to the next higher power of two, which is 2,097,150 (2**21, or 0x200000). Adding a guaranteed rate less than 150 Mbps does not alter the result, as the larger of the two rate pairs is used. Given that CIR is always less than or equal to PIR, when both are set the PIR is used to compute the default burst size.

Choosing the Actual Burst Size. When no burst size is specified, the system uses a default burst for a given rate-pair that is based on 100 milliseconds, as described previously. Otherwise, when a burst size is configured, the system uses the following algorithm to choose the actual burst:

- If only one of the rate pair shapers is configured with a burst size, use its configured burst size.
- If both rate pair shapers are configured with a burst size, use the lesser of the two burst sizes.

- Round the selected burst size *down* to the nearest power of two; note this differs from the case of computing a default burst rate, where the value is rounded up to nearest power of two.
- Calculate a platform-dependant minimum burst size.
- Compare configured burst to the platform specific minimum burst size, select the larger of the two as the actual burst rate. If the user-configured value exceeds the platform specific maximum, then select the platform maximum as the burst rate.

> There is no indication on the RE's CLI as to whether the user-configured or platform-specific minimum or maximum burst size has been programmed. PFE level debug commands are needed to see the actual burst size programmed.

Burst Size Example. The best way to get all this straights is through a concrete example. R4 is confirmed to have a Q style MPC:

```
[edit]
jnpr@R4# run show chassis hardware | match fpc
FPC 2            REV 15   750-031088   YR7240           MPC Type 2 3D Q
```

And the configuration is set to shape the IFD at 500 Mbps with a user-configured shaper burst size of 32kB, which is confirmed in the operational mode CLI command:

```
[edit]
jnpr@R4# show class-of-service traffic-control-profiles tc-ifd-500m
shaping-rate 500m burst-size 40k;
overhead-accounting bytes -20;
shaping-rate-priority-high 200m;

[edit]
jnpr@R4# run show class-of-service traffic-control-profile tc-ifd-500m
Traffic control profile: tc-ifd-500m, Index: 17194
  Shaping rate: 500000000
  Shaping rate burst: 40000 bytes
  Shaping rate priority high: 200000000
  Scheduler map: <default>
```

To compute the actual burst size, the larger of the burst sizes configured for the PIR/CIR pairing is rounded down to the nearest power of 2. Here only PIR is specified, so its burst size is used:

40,000 rounded down to nearest power of 2: 32,000 (2^5).

Then a minimum burst size is computed based on the hardware type (Q versus EQ) and the node's position, both of which influences the minimum burst value in milliseconds. In this example, the 1.837 milliseconds value is used, given this is a L1 node on a Q type MPC:

500 Mbps * 1.837 ms/8000 = 114,812.5 bytes.

The resulting value is then rounded up to the nearest power of two:

114,812.5 rounded up to nearest power of 2: 131,072 (2^17)

Select the larger of the two, which in this case is the system minimum of 131,072 bytes. PFE shell commands are currently needed to confirm the actual burst value:

```
NPC2(R4 vty)# sho cos scheduler-hierarchy

class-of-service EGRESS scheduler hierarchy - rates in kbps
---------------------------------------------------------------------------------
                                      shaping guarntd delaybf  excess
interface name                 index    rate    rate    rate    rate      other
---------------------------- -----  ------- ------- ------- ------- -------------
xe-2/0/0                       148        0       0       0       0
xe-2/0/1                       149        0       0       0       0
xe-2/1/0                       150        0       0       0       0
xe-2/1/1                       151        0       0       0       0
xe-2/2/0                       152   500000       0       0       0
  iflset_premium                25    30000   20000       0       0
    xe-2/2/0.200               335     3000    2000       0       0
      q 0 - pri 0/0          20205        0    1000       0     35%
      q 1 - pri 0/0          20205        0    1000       0      5%
      q 2 - pri 0/0          20205        0    1000       0     10%
      q 3 - pri 3/1          20205        0    1000     10%      5%
      q 4 - pri 0/0          20205        0    1000       0     30%
      q 5 - pri 4/0          20205        0    1000   25000      0% exact
      q 6 - pri 0/0          20205        0    1000       0     15%
      q 7 - pri 2/5          20205        0       0       0      0%


NPC2(R4 vty)# sho cos halp ifd 152

--------------------------------------------------------------------------------
IFD name: xe-2/2/0   (Index 152)
    QX chip id: 1
    QX chip L1 index: 1
    QX chip dummy L2 index: 1
    QX chip dummy L3 index: 3
    QX chip base Q index: 24
Queue    State         Max       Guaranteed   Burst  Weight Priorities Drop-Rules
Index                  rate         rate      size            G    E   Wred  Tail
------ ----------- ----------- ------------ ------- ------ ---------- ----------
    24  Configured   500000000            0 2097152    950   GL   EL     4    82
    25  Configured   500000000            0 2097152      1   GL   EL     0   255
    26  Configured   500000000            0 2097152      1   GL   EL     0   255
    27  Configured   500000000            0 2097152     50   GL   EL     4    25
    28  Configured   500000000            0 2097152      1   GL   EL     0   255
    29  Configured   500000000            0 2097152      1   GL   EL     0   255
    30  Configured   500000000            0 2097152      1   GL   EL     0   255
    31  Configured   500000000            0 2097152      1   GL   EL     0


NPC2(R4 vty)# show qxchip 1 l1 1
L1 node configuration   : 1
        state           : Configured
```

```
child_l2_nodes  : 2
config_cache    : 21052000
rate_scale_id   : 0
gh_rate         : 200000000, burst-exp 18 (262144 bytes scaled by 16)
gm_rate         : 0
gl_rate         : 0
eh_rate         : 0
el_rate         : 0
max_rate        : 500000000
cfg_burst_size  : 32768 bytes
burst_exp       : 13 (8192 bytes scaled by 16)
byte_adjust     : 4
cell_mode       : off
pkt_adjust      : 0
```

The configured and actual burst size values are shown. Note the configured has been rounded down. Multiplying 8,192 by the scale factor shown yields the actual burst value, which is 131,072, as expected.

Shapers and Delay Buffers

Delay buffers are used by shapers, either CIR or PIR, to absorb bursts so that the output of the shaper is able to maintain some degree of smoothness around the shaped rate. While adding a “bigger buffer” is a genuine fix for some issues, you must consider that a bigger buffer trades loss and shaper accuracy for increased latency. Some applications would prefer loss due to a small buffer, rather than being delivered late in the eyes of a real-time application, where they may be seen as causing more harm than good.

The delay buffer can be increased from the default 100 ms to 200 ms of the port speed and can also be oversubscribed beyond 200 milliseconds using the `delay-buffer-rate` configuration at the port IFD level via a TCP. The maximum delay buffer varies by MPC type and is currently limited to 500 milliseconds of shaped rate, which for a 1 Gbps interface is equivalent to 500 Mb or 62.5 MB. Unlike the IQ2 hardware, Trio does not use dynamic buffer allocations. There is no concept of MAD (Memory Allocation Dynamic) in Trio.

> The 16x10GE MPC uses four Trio PFEs. Each PFE has a 5 Gb delay buffer, which provides each of the 16 ports (4 ports per PFE) with 100 milliseconds of delay bandwidth buffer. The remaining 100 milliseconds of delay buffer can be allocated among the four ports on each PFE using CLI configuration.

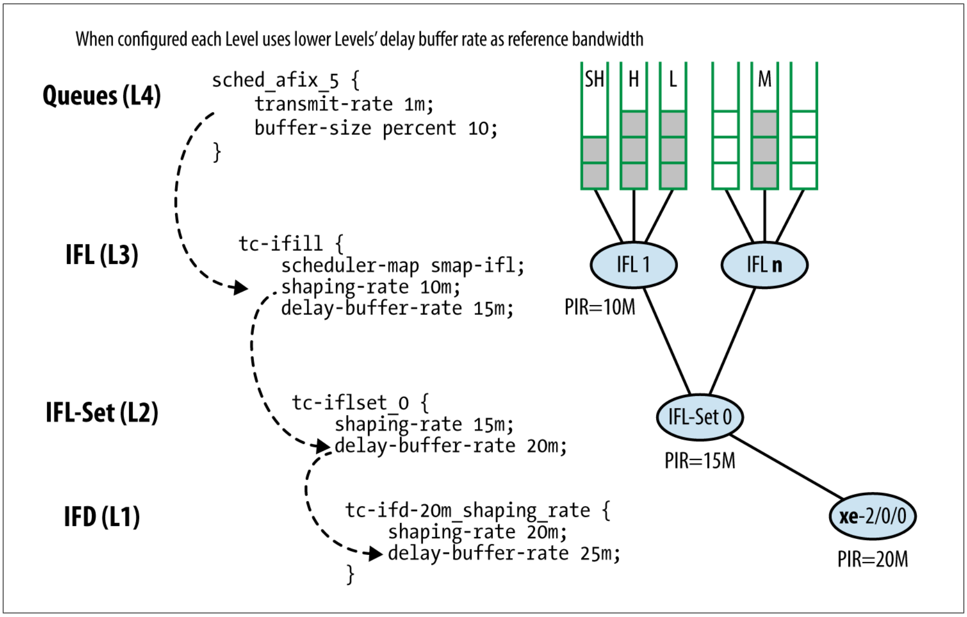
By default, the delay-buffer calculation is based on the guaranteed rate, the shaping rate if no guaranteed rate is configured. You can alter the default behavior with the `delay-buffer-rate` parameter in a TCP definition. This parameter overrides the shaping rate as the basis for the delay buffer calculation. If any logical interface has a configured guaranteed rate, all other logical interfaces on that port that do not have a guaranteed rate configured receive a delay buffer rate of 0. This is because the absence of a guaranteed rate configuration corresponds to a guaranteed rate of 0 and, consequently, a delay buffer rate of 0.

> When an interface is oversubscribed and you do not specify a shaping rate or a guaranteed rate, or a delay buffer rate, the node receives a minimal delay buffer rate and minimal bandwidth equal to two MTU-sized packets.

You can configure a rate for the delay buffer that is higher than the guaranteed rate. This can be useful when the traffic flow might not require much bandwidth in general, but in some cases traffic can be bursty and therefore needs a large buffer. Configuring large buffers on relatively slow-speed links can cause packet aging. To help prevent this problem, the software requires that the sum of the delay buffer rates be less than or equal to the port speed. This restriction does not eliminate the possibility of packet aging, so you should be cautious when using the `delay-buffer-rate` statement. Though some amount of extra buffering might be desirable for burst absorption, delay buffer rates should not far exceed the service rate of the logical interface. If you configure delay buffer rates so that the sum exceeds the port speed, the configured delay buffer rate is not implemented for the last logical interface that you configure. Instead, that logical interface receives a delay buffer rate of 0, and a warning message is displayed in the CLI. If bandwidth becomes available (because another logical interface is deleted or deactivated, or the port speed is increased), the configured `delay-buffer-rate` is reevaluated and implemented if possible. If the guaranteed rate of a logical interface cannot be implemented, that logical interface receives a delay buffer rate of 0, even if the configured delay buffer rate is within the interface speed. If at a later time the guaranteed rate of the logical interface can be met, the configured delay buffer rate is reevaluated; if the delay buffer rate is within the remaining bandwidth, it is implemented.

Delay Buffer Rate and the H-CoS Hierarchy. You configure the `delay-buffer-rate` parameter in TCPs that are attached scheduling nodes in the H-CoS model. In port-level queuing, the speed (physical or shaped) of the IFD is implicitly known and is used as the basis for queue-level delay buffers. In H-CoS, the bandwidth of a VLAN/IFL cannot be implicitly derived and requires explicit configuration for delay bandwidth calculations. If a delay buffer value is not explicitly configured, the bandwidth is based on the specified CIR, or when no CIR is specified the PIR.

Generally speaking, the delay buffer rate setting at one level of the hierarchy becomes the reference bandwidth used at the next higher layer, and the sum of reference bandwidth cannot exceed the value used at a lower layer. Figure 5-13 shows these concepts.



```
sched_afix_5 {
    transmit-rate 1m;
    buffer-size percent 10;
}
```

```
tc-ifill {
    scheduler-map smap-ifl;
    shaping-rate 10m;
    delay-buffer-rate 15m;
```

```
tc-iflset_0 {
    shaping-rate 15m;
    delay-buffer-rate 20m;
```

```
tc-ifd-20m_shaping_rate {
    shaping-rate 20m;
    delay-buffer-rate 25m;
}
```

Figure 5-13. Delay Buffer Rate and H-CoS.

Figure 5-13 shows an example of explicitly setting delay buffer rate to be 5 Mb (bits) higher than that level's shaping rate, thereby overriding the default of using CIR when configured, else PIR. Note how the queue-level scheduler setting for `buffer-size` uses the IFL layer's delay buffer rate as its reference, thereby getting 10% of 15 Mb in this example. In turn, IFL 1 then feeds into an IFL-Set at level 2, which again has an explicit delay buffer rate configured, with the result being that the reference bandwidth for the IFL 1 is 25 MB rather than the 15 Mb shaped rate. Also note how the sum of delay buffers at Layers 4 through 2 are less than the delay buffer at level 1, the IFD. For predictable operation, this should be true at all levels of the hierarchy.

Sharing Excess Bandwidth

Historically, with Junos CoS, once all nodes/queues receive their guaranteed rate, the remaining excess bandwidth is divided among them equally. As this method of sharing excess bandwidth was not always desired, Trio platforms provide additional control. Specifically, the `excess-rate` option used in a TCP for application to a scheduler node, combined with the `excess-rate` and the `excess-priority` options that are configured at the scheduler level for application to queues.

The excess-bandwidth-share(equal | proportional) option, applied at the [edit class-of-service interface (interface-set | interface)] hierarchy is used for queuing DPCs only. The option is not applicable to Trio MPCs.

When applied to a scheduling node, you control the sharing of excess bandwidth among different users, for example how much excess bandwidth a given IFL can use when you configure excess rate at a level 2 IFL-Set node. When applying at the queue level, you can combine excess-rate/excess-priority with a shaping rate to control the sharing of excess bandwidth among services from a single user (i.e., between queues attached to a specific C-VLAN/IFL). Typically, users configure excess sharing at both scheduler and queue levels for control over both node and queue levels of excess sharing.

For both nodes and queues, you specify excess rate as either a percentage or a proportion. Percent values are from 1 to 100, and you can configure a total percentage that exceeds 100%. Proportional values range from 1 to 1,000. By default, excess bandwidth is shared equally among siblings in the scheduler hierarchy.

It's a good practice to configure either a percentage or a proportion of the excess bandwidth for all schedulers with the same parent in the H-CoS hierarchy; try not to mix percentages and proportions as things are hard enough to predict without the added complexity. In addition, when using an approach that is based on percentages, try and make them sum to 100%. For example, if you have two IFLs in a set, configure interface xe-1/1/1.0 with 20% of the excess bandwidth, and configure interface xe-1/1/1.1 with 80% of the excess bandwidth.

Scheduler Nodes. Shaper and schedulers are applied to level 2 or 3 scheduler nodes through a TCP. You can configure excess bandwidth sharing within a TCP as shown:

```
jnpr@R1-RE0# set class-of-service traffic-control-profiles test excess-rate?
Possible completions:
> excess-rate          Excess bandwidth sharing proportion
> excess-rate-high     Excess bandwidth sharing for excess-high priority
> excess-rate-low      Excess bandwidth sharing for excess-low priority
{master}[edit]
```

The excess-rate keyword value specified applies to both excess priority levels. Starting with the v11.4 release, you can specify different excess rates based on the priority. You use the high and low excess options when you need to share excess bandwidth differently based on traffic type. For example, you might have all users send BE at excess priority low with a 1:1 ratio, meaning any BE traffic sent as excess results in the scheduler sending one packet from each user in a WRR manner, thus no differentiation is provided for BE, or for any other traffic sent at excess low for that matter. To provide a gold level of service, the operator can set business class users so that their schedulers demote to excess high priority, while normal users have all their schedulers set for excess low. You can set excess high to be shared at a different ratio, for example, 4:1, to provide the business class users four times the share of excess bandwidth for that traffic type.

When `excess-rate` is specified, you cannot specify an `excess-rate-high` or `excess-rate-low` as the CLI prevents configuring `excess-rate` along with either of these two attributes.

Queues. In the case of queues, you can also define the priority level for excess traffic.

```
jnpr@R1-RE0# set class-of-service schedulers test excess-priority ?
Possible completions:
  high
  low
  medium-high
  medium-low
  none
```

> While the CLI offers all priority levels, in the current release only `high`, `low`, and `none` are valid. Specifying an unsupported option will result in the default setting that is based on the queue's normal priority.

The configuration of `excess-rate` or `excess-priority` is prohibited for a queue when:

Shaping with the `transmit-rate exact` statement, because in this case the shaping rate equals the transmit rate, which means the queue can never operate in the excess region.

The scheduling priority is set to `strict-high`, which means the queue's transmit rate is set to equal the interface bandwidth, which once again means can never operate in excess region.

> In testing the v11.4R1 Junos release, it was found that excess-priority none could be combined with strict-high and it appeared to take effect. Documentation PR 782534 was filed to sort things out. The expected commit error was encountered when `excess-rate` was combined with `strict-high`.

Excess None. Starting in the v11.4 release, you can prevent priority demotion at the queue level by specifying an excess priority level of none. You can only use this option when combined with a scheduler priority of high or medium; a commit error is generated when combined with priority low, as this guaranteed level is associated with G-Rate-based priority demotion at scheduler nodes and therefore must remain eligible for demotion, as described previously. Unless rate limited, once a queue hits its transmit rate, it switches from its normal priority to its configured excess priority, and can then continue to send, being capped by its shaping rate if specified, else the shaping rate of the

IFL, else the shaping rate of IFL-Set, else the shaping rate of the IFD; unless, of course, it runs out of excess bandwidth first.

If a shaping rate is not specified, and there is no other traffic contention from other queues, then even a queue with a small transmit rate is able to send at the interface rate, albeit with the majority of its traffic in the excess region in such a case. This behavior is considered a feature, but some users found that priority demotion later muddied their plans to perform priority-based shaping on a traffic class aggregate, as some of the traffic they expected to shape at priority value x was demoted and allowed to bypass the planned priority-x shaper. The fix to this problem is to specify `excess-rate none` for queues at priority x.

Combining excess none with a low transmit rate is an excellent way to starve a queue; this is considered a feature, perhaps to enable the offering of a penalty box CoS level reserved for users that don't pay their bills on time. To help avoid unwanted starvation, a safety check is in place to generate an error when excess priority none is set on a scheduler that has a transmit rate expressed as a percent when the parent level 2 IFL scheduling node has a guaranteed rate (CIR) of zero.

Shaping with Exact Versus Excess Priority None

Users often ask, "What is the difference between shaping a queue using exact and preventing demotion through excess priority none?" It's a great question, and one with a somewhat convoluted answer. Typically, you will see the same behavior with the two options, as both shape the queue to the configured transmit rate. However, the fact that the later prevents any and all demotion, while the former does not, means there can be a difference, especially if you are overbooking G-Rates, using GL which is eligible for demotion, or performing per priority shaping.

A queue with `exact` that is within its transmit rate, if at GL, can be demoted into excess at L2/L3, nodes, or it can be demoted at any node if the traffic is found to exceed a per priority shaper. Note that a queue with `exact` that is within its transmit rate is not demoted as a function of G-Rate if it's set to GH or GM priority.

In contrast, a queue with excess priority none can never ever be demoted; this is why you cannot combine the excess none option with the GL priority, as GL must remain demotable to facilitate G-Rate handing at L2/L3 nodes.

So, in the case of a per priority shaper that wants to demote, an excess none queue that is within its transmit rate will see back pressure, while a queue set for exact may get demoted (even though the queue is within its transmit rate).

You can create a scheduler with no transmit rate to blackhole traffic by omitting a transmit rate and specifying an excess priority of none:

```
{master}[edit]
jnpr@R1-RE0# show class-of-service schedulers sched_null
priority medium-high;
excess-priority none;
```

Excess Handling Defaults. The default behavior for excess rate varies based on interface mode of PIR or PIR/CIR, and whether an excess rate has been explicitly configured:

PIR Mode (none of the IFLs/IFL-Sets have `guaranteed-rate` *configured)*
: The excess rate is based on shaping rate, by default an excess rate of 0% with weight proportional to queue transmit ratio.

PIR/CIR Mode (at least one IFL/IFL-Set has `guaranteed-rate` *configured)*
: The excess rate is based on guaranteed rate, else shaping rate. The default excess rate is 0% with weight proportional to queue transmit ratio.

Excess-Rate Mode (at least one IFL/IFL-Set has `excess-rate` *configured)*
: When explicitly configured, excess rate is equal to configured value, else excess rate is 0% with weight proportional to configured excess rate, else 0.

The last bit bears repeating. In both PIR and CIR modes, if you assign an excess rate to a queue, the interface enters the excess rate mode. Any queue that does not have an `excess-rate` assigned gets a 0 weight for excess sharing, even if that queue is at high priority with a have a guaranteed rate!

> The default excess sharing has all queues at 0% excess rate with a weighting that is based on the ratio of the queue transmit rates. The result is that a queue with two times the transmit rate of another also gets two times the excess bandwidth.
>
> If you plan to deviate from the default excess rate computation, you should configure all queues with an excess rate, even those queues that you actually want to have 0 excess weighting. Having a queues go from an excess weight that is based on queue transmit ratios to an excess weight of 0 can be a big change, and not one that is typically anticipated.

Excess Rate and PIR Interface Mode. Given that the `excess-rate` statement is intended to distribute excess bandwidth once scheduling nodes reach their guaranteed rate and none of the scheduling nodes have guaranteed rate configured when in PIR mode, it can be argued that allowing an excess rate configuration doesn't make sense in PIR mode. However, due to customer feedback, the rules were relaxed, and as of v11.4 you can now specify an excess rate for interfaces in PIR mode. As a result of the change, the commit is no longer blocked and the configured excess rates are honored. As with the CIR model, queues drop the priority into an excess region when sending above the configured transmit rate with excess shared based on the ratio of the configured excess rates.

Excess Sharing Example. Figure 5-14 shows an example of excess sharing in a PIR/CIR H-CoS environment.

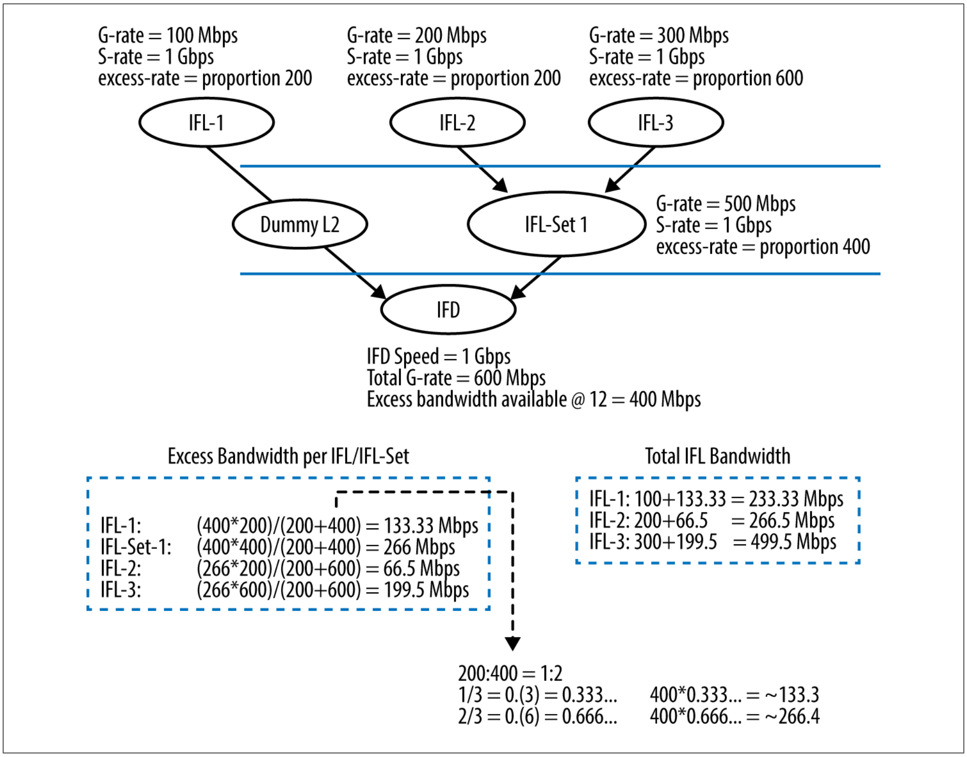

G-rate = 100 Mbps
S-rate = 1 Gbps
excess-rate = proportion 200
G-rate = 200 Mbps
S-rate = 1 Gbps
excess-rate = proportion 200
G-rate = 300 Mbps
S-rate = 1 Gbps
excess-rate = proportion 600
IFL-1
IFL-2
IFL-3
Dummy L2
IFL-Set 1
G-rate = 500 Mbps
S-rate = 1 Gbps
excess-rate = proportion 400
IFD
IFD Speed = 1 Gbps
Total G-rate = 600 Mbps
Excess bandwidth available @ 12 = 400 Mbps
Excess Bandwidth per IFL/IFL-Set
IFL-1: (400*200)/(200+400) = 133.33 Mbps
IFL-Set-1: (400*400)/(200+400) = 266 Mbps
IFL-2: (266*200)/(200+600) = 66.5 Mbps
IFL-3: (266*600)/(200+600) = 199.5 Mbps
Total IFL Bandwidth
IFL-1: 100+133.33 = 233.33 Mbps
IFL-2: 200+66.5 = 266.5 Mbps
IFL-3: 300+199.5 = 499.5 Mbps
200:400 = 1:2
1/3 = 0.(3) = 0.333...
2/3 = 0.(6) = 0.666...
400*0.333... = ~133.3
400*0.666... = ~266.4

Figure 5-14. Excess Bandwidth Sharing—IFL Level.

Here, the focus is on IFL- and IFL-Set-level sharing. We begin at the IFD level, which is shown to be a 1 Gbps interface. The sum of the G-Rates assigned to the IFLs, and to IFL-Set 1, sum to 600 Mbps, thus leaving a total of 400 Mbps remaining for excess bandwidth usage. In this example, the default behavior of using the IFL transmit ratios to set the proportion for excess sharing.

The first round of sharing occurs at L2, which given the dummy L2 node shown for IFL 1 means the first division is between IFL 1 and IFL-Set 1. In this case, the former has a proportion of 200 while the latter has 400, thus leading to the math shown. Excess rates are set as a proportional value from 1 to 1,000, and so this slide uses realistic excess weighting values. In this case, the 200/400 proportion can be simplified to a 1:2 ratio. In decimals, these can be rounded to 0.3333 and 0.6666, respectively; the alternate form of math is also shown in Figure 5-14. Multiplying those values by the available excess share (400 M * 0.3333 = ~ 133.3 M) yields the same numbers on the slide, but requires the decimal conversion.

The result is IFL 1 gets 133.3 Mbps of excess bandwidth, while the IFL-Set nets two times the excess bandwidth, or 266 Mbps, which is expected given it has twice the excess rate proportion. A similar calculation is now performed for the two IFLs shown belonging to the IFL-Set, where the two IFLs are now contending for the remaining 266 Mbps. Their proportions are 200 and 600, respectively, leading to the math shown and the resulting 66.5 Mbps for IFL 2 and the 199.5 Mbps for IFL 3.

The final result is shown in the right-hand table, where the IFL G-Rate is added to the calculated excess rates to derive the expected maximum rate for each IFL. Figure 5-15 builds on the current state of affairs by extending the calculation into level 4, the domain of queues.

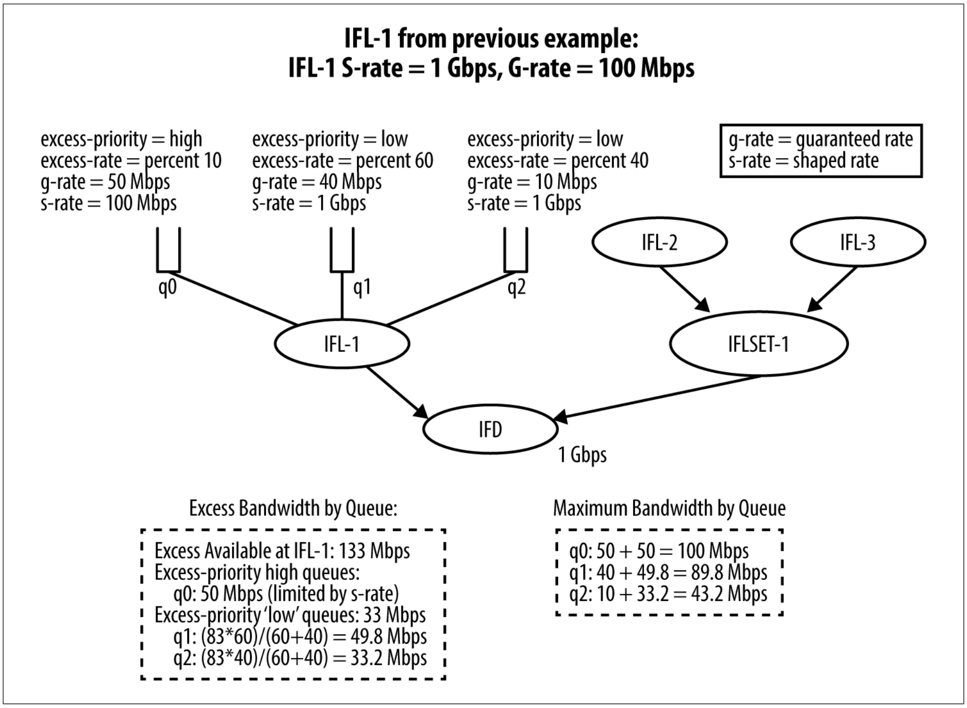

IFL-1 from previous example:
IFL-1 S-rate = 1 Gbps, G-rate = 100 Mbps
excess-priority = high
excess-rate = percent 10
g-rate = 50 Mbps
s-rate = 100 Mbps
excess-priority = low
excess-rate = percent 60
g-rate = 40 Mbps
s-rate = 1 Gbps
excess-priority = low
excess-rate = percent 40
g-rate = 10 Mbps
s-rate = 1 Gbps
g-rate = guaranteed rate
s-rate = shaped rate
q0
q1
q2
IFL-2
IFL-3
IFL-1
IFLSET-1
IFD
1 Gbps
Excess Bandwidth by Queue:
Excess Available at IFL-1: 133 Mbps
Excess-priority high queues:
q0: 50 Mbps (limited by s-rate)
Excess-priority 'low' queues: 33 Mbps
q1: (83*60)/(60+40) = 49.8 Mbps
q2: (83*40)/(60+40) = 33.2 Mbps
Maximum Bandwidth by Queue
q0: 50 + 50 = 100 Mbps
q1: 40 + 49.8 = 89.8 Mbps
q2: 10 + 33.2 = 43.2 Mbps

Figure 5-15. Excess Bandwidth Sharing—Queue Level.

In this case, we focus on the queues assigned to IFL 1, which recall had a 1G shaping rate, a G-Rate of 100 M, along with a computed 133.3 Mbps of excess bandwidth capacity, yielding a guaranteed throughput potential for the IFL of 233.3 to be divided among its queues. The IFL is shown with three queues (0 to 2) with transmit rates (G-Rate) of 50 Mbps, 40 Mbps, and 10 Mbps, respectively. The queues are set with excess rate specified as a percentage, in this case 10%, 60%, and 40%, respectively, along with an excess priority of EH, EL, and EL, respectively. Given that G-Rates are not overbooked, we can assume that all three queues can send at their transmit rates simultaneously with up to 133.3 Mbps of excess available for division among them.

The table on the left computes the excess bandwidth expected for each queue, as a function of priority, which in Trio scheduling is absolute. Note that the `excess-rate` parameter sets a minimal fair share, but is in itself *not an actual limit on excess bandwidth usage*. Thus, Q0 is therefore limited only by its shaping rate of 100 Mbps, and not the sum of its G-Rate + excess rate, and therefore is expected to reach the full 100 Mbps shaping rate. This leaves 83 Mbps for queues 1 and 2, which being at the same priority split the remaining excess based on their excess share percentages, a 60/40 split (3:2 ratio) in this case, results in 49.8 Mbps to Q1 and the remaining 33.2 Mbps going to Q2. The table on the right adds the queue G-Rate to its computed maximum excess rate to derive the maximum expected throughput for each queue when all are fully loaded.

If some queues are not using their bandwidth share, other queues can achieve high throughputs, but as always the queue's maximum bandwidth is limited by the shaping rate. If a shaping rate is not set at the queue level, then the queue limit becomes the shaping rate of the IFL, IFL-Set, or IFD, respectively. If no shaping is performed, IFD physical speed is the queue bandwidth limit.

Priority-Based Shaping

Referring back to the H-CoS reference hierarchy, shown in Figure 5-7, you can see that Trio H-CoS supports shaping traffic based on its priority at L1, L2, and L3 scheduler nodes. You can define a shaping rate for each of the supported priorities to control aggregate rates on a per priority basis. You can combine priority-based shaping with queue-, IFL-, and IFL-Set-level shaping to exercise control over the maximum rate of a user (IFL), and an IFL-Set for all priorities, and then shape per priority at the IFD level to place an aggregate cap on a per priority basis for all users and IFL/IFL-Sets; this cap might be less than the sum of the IFL-Set shaping or G-Rates, such that statistical multiplexing comes into play for certain priorities of traffic while also allowing network capacity planning based on a fix aggregate that is independent of total user IFL and IFL-Set counts on the interface.

On possible usage might be to cap overall high-priority VoIP traffic to help ensure that it's not starving other priority levels due to excess call volume, perhaps because someone fails to factor Mother's Day call volume.

Priority shaping is configured within a TCP; the options are shown:

```
[edit]
jnpr@R1# set class-of-service traffic-control-profiles test shaping-rate-e?
Possible completions:
> shaping-rate-excess-high  Shaping rate for excess high traffic
> shaping-rate-excess-low  Shaping rate for excess low traffic
[edit]
jnpr@R1# set class-of-service traffic-control-profiles test shaping-rate-p?
Possible completions:
> shaping-rate-priority-high  Shaping rate for high priority traffic
> shaping-rate-priority-low  Shaping rate for low priority traffic
> shaping-rate-priority-medium  Shaping rate for medium priority traffic
[edit]
jnpr@R1# set class-of-service traffic-control-profiles test shaping-rate-p
```

Here the IFD-level TCP that shapes all traffic to 500 Mbps is modified to shape priority medium at 10 Mbps and applied to the IFD level:

```
{master}[edit]
jnpr@R1-RE0# show class-of-service traffic-control-profiles 500m_shaping_rate
shaping-rate 500m;
shaping-rate-priority-medium 10m;

jnpr@R1-RE0# show class-of-service interfaces xe-2/0/0 unit 0
output-traffic-control-profile 500m_shaping_rate;
classifiers {
. . . .
```

And the result is confirmed, both in the CLI and in the MPC itself:

```
{master}[edit]
jnpr@R1-RE0# run show class-of-service traffic-control-profile 500m_shaping_rate
Traffic control profile: 500m_shaping_rate, Index: 54969
  Shaping rate: 500000000
  Shaping rate priority medium: 10000000
  Scheduler map: <default>

NPC2(R1-RE0 vty)# sho cos ifd-per-priority-shaping-rates

EGRESS IFD Per-priority shaping rates, in kbps
                                     per-priority shaping-rates (in kbps)
       ----------------------------- ---------------------------------------
Ifd     Shaping   Guarantd  DelayBuf    GH      GM      GL     EH      EL
Index     Rate      Rate      Rate     Rate    Rate    Rate    Rate    Rate
------ --------- --------- --------- ------- ------- ------- ------- -------
   148    500000    500000    500000       0   10000       0       0       0
   149         0  10000000  10000000       0       0       0       0       0
   150         0  10000000  10000000       0       0       0       0       0
   151         0  10000000  10000000       0       0       0       0       0
   152         0  10000000  10000000       0       0       0       0       0
   153         0  10000000  10000000       0       0       0       0       0
   154         0  10000000  10000000       0       0       0       0       0
   155         0  10000000  10000000       0       0       0       0       0
```

Per priority shaping does not occur at the queue level, but you can control what priority a queue uses for traffic within and in excess of its transmit rate through the queue-level excess priority setting, as a function of priority inheritance. You can shape overall queue bandwidth using the `shaping-rate` statement, but this is not on a priority basis.

Why Does Priority-Based Policing Not Limit Queue Bandwidth?

A common question to be sure. Imagine you have a queue set to priority high with a rate limit of 1 Mbps. Below that queue, you have an IFL with a shaping rate priority high 500 kbps statement. You expect that upon commit the queue will fall back to the IFL priority shaping rate of 0.5 Mbps, but instead it remains at 1 Mbps. Why?

The answer is priority demotion at scheduler nodes, here at level 3. When the incoming rate of the GH traffic exceeds the priority shaper, the node demotes the traffic to the queue's excess priority level. Granted, the traffic is no longer at the same priority, but if there's no congestion it's delivered. You can set the queue to `excess-priority none` to prevent this behavior, and have per priority policers impose a limit on queue throughput. Note that `excess-priority none` cannot be used on queues set for the default GL priority.

Fabric CoS

The concept of fabric CoS was mentioned in a previous section on "intelligent oversubscription." In addition, the use of the switch fabric to move traffic from ingress to egress MPC, using a request/grant mechanism, was discussed in Chapter 1 "Trio Hardware Architecture." Here, we focus on how CoS configuration is used to mark selected traffic types as high priority to ensure they have minimum impact should any fabric congestion occur.

The default setting of switch fabric priority varies by interface mode. In port mode, any schedulers that use high priority automatically map that FC to a high fabric priority. In H-CoS mode, all FCs default to normal fabric priority, regardless of the associated scheduling priority. To alter, you must explicitly set a high priority at the `[edit class-of-service forwarding-classes class<name>]` hierarchy:

```
jnpr@R1-RE0# set forwarding-classes class ef priority ?
Possible completions:
  high                 High fabric priority
  low                  Low fabric priority
{master}[edit class-of-service]
jnpr@R1-RE0# set forwarding-classes class ef priority high

{master}[edit class-of-service]
jnpr@R1-RE0# commit
```

The priority change is confirmed:

```
{master}[edit class-of-service]
jnpr@R1-RE0# run show class-of-service forwarding-class
Forwarding   ID    Queue  Restricted   Fabric    Policing    SPU
class                     queue        priority  priority    priority
  be         0     0          0         low       normal      low
  af1x       1     1          1         low       normal      low
  af2x       2     2          2         low       normal      low
  nc         3     3          3         low       normal      low
  af4x       4     4          0         low       normal      low
  ef         5     5          1         high      premium     low
  af3x       6     6          2         low       normal      low
  null       7     7          3         low       normal      low
```

Note that only the EF class has an altered fabric priority, despite both the EF and NC schedulers having the same high priority:

```
{master}[edit class-of-service]
jnpr@R1-RE0# show schedulers sched_nc
transmit-rate 500k;
```

```
    priority high;

{master}[edit class-of-service]
jnpr@R1-RE0# show schedulers sched_ef_50
transmit-rate {
    2m;
    rate-limit;
}
buffer-size temporal 25k;
priority strict-high;
```

Given that FC to switch fabric priority mapping defaults vary by an interface's CoS mode, the best practice is to explicitly set the desired fabric priority, a method that yields predictable operation in all CoS modes. If desired, you can combined WRED drop profiles for each fabric priority using a scheduler map. This example uses the default RED profile for high fabric priority while using a custom drop profiles for low fabric priority traffic:

```
{master}[edit]
jnpr@R1-RE0# show class-of-service fabric
scheduler-map {
    priority low scheduler sched_fab_high;
}
```

The linked schedulers only support drop profiles because concepts such as scheduler priority and transmit rate have no applicability to a fabric scheduler; the traffic's fabric priority is set as described previously, outside of the scheduler:

```
jnpr@R1-RE0# show class-of-service schedulers sched_fab_high
drop-profile-map loss-priority high protocol any drop-profile dp-fab-high;
drop-profile-map loss-priority low protocol any drop-profile dp-fab-low;
```

And the customer WRED fabric profile is confirmed for low fabric priority traffic:

```
NPC2(R1-RE0 vty)# sho cos fabric scheduling-policy
 Fabric    plp/tcp  plp/tcp  plp/tcp  plp/tcp
 Priority   (0/0)    (1/0)    (0/1)    (1/1)
 -------- -------- -------- -------- --------
 low        64745    48733    64745    48733
 high           1        1        1        1
```

The result is more aggressive drops for low fabric priority while the default WRED profile does not begin discarding high fabric priority until 100%.

Control CoS on Host-Generated Traffic

You can modify the default queue assignment (forwarding class) and DSCP bits used in the ToS field of packets generated by the RE using the `host-outbound-traffic` statement under the `class-of-service hierarchy`, or through a firewall filter applied in the output direction of the loopback interface.

Default Routing Engine CoS

By default, the forwarding class (queue) and packet loss priority (PLP) bits are set according to the values given in the default DSCP Classifier. TCP-related packets, such as BGP or LDP sessions, first use queue 0 (BE) and then fall back to use queue 3 (network control) only when performing a retransmission.

The default outgoing queue and FC selection for selected RE traffic is shown in Table 5-10. The complete list can be found at *http://www.juniper.net/techpubs/en_US/junos11.4/topics/reference/general/hw-cos-default-re-queues-reference-cos-config-guide.html*.

Table 5-10. Default Queue Mappings for RE-Generated Traffic.

Protocol	Queue
ATM OAM	Queue 3
Bidirectional Forwarding Detection (BFD) Protocol	Queue 3
BGP/BGP Retransmission	Queue 0/Queue 3
Cisco High-Level Data Link Control (HDLC)	Queue 3
FTP	Queue 0
IS-IS	Queue 3
IGMP query/report	Queue 3/Queue 0
IPv6 Neighbor Discovery	Queue 3
IPv6 Router Advertisement	Queue 0
LDP UDP hellos (neighbor discovery)	Queue 0
LDP TCP session data/retransmission	Queue 0/Queue 3
Link Aggregation Control Protocol (LACP)	Queue 3
Open Shortest Path First (OSPF)	Queue 3
PPP (Point-to-Point Protocol	Queue 3
PIM (Protocol Independent Multicast)	Queue 3
Real-time performance monitoring (RPM) probe	Queue 3
Resource Reservation Protocol (RSVP)	Queue 3
Simple Network Management Protocol (SNMP)	Queue 0
SSH/Telnet	Queue 0
VRRP	Queue 3

The default ToS markings for RE traffic can vary by type.

Routing Protocols

Protocols like OSPF, RSVP, IS-IS, PIM, LDP, RIP, and so on use the CS6 IP precedence value of 110. It may seem odd they are not sent with CS7/111, but with a default IP precedence classifier Junos sees a set LSB as an indicator for high drop probability, making CS6 more reliable than CS7 should congestion occur.

Management Traffic

Traffic such as NTP, DNS, Telnet, and SSH use routine precedence 000.

Variable, based on Request

ICMP and SNMP traffic uses variable settings. When locally generated, the default 000 is used, but when responding to a request the replies are set with the same ToS marking as in the request.

To change the default queue and DSCP bits for RE sources traffic, include the `host-outbound-traffic` statement at the `[edit class-of-service]` hierarchy level. Changing the defaults for RE-sourced traffic does not affect transit or incoming traffic, and the changes apply to all packets relating to Layer 3 and Layer 2 protocols, but not MPLS EXP bits or IEEE 802.1p bits. This feature applies to all application-level traffic such as FTP or ping operations as well. The following notes regarding EXP and VLAN tagging IEEE 802.1p should be kept in mind:

For all packets sent to queue 3 over a VLAN-tagged interface, the software sets the 802.1p bit to 110.
For IPv4 and IPv6 packets, the software copies the IP type-of-service (ToS) value into the 802.1p field independently of which queue the packets are sent out.
For MPLS packets, the software copies the EXP bits into the 802.1p field.

As with any classification function, care must be taken to ensure the queue selected is properly configured and scheduled on all interfaces. It is always good practice to leave queue 3 associated with the network control forwarding class and to use extreme caution when placing any other types of traffic into this queue. Starving network control traffic never leads to a desirable end, unless your goal is networkwide disruption of services.

This example places all routing engine-sourced traffic into queue 3 (network control) with a DSCP code point value of 101010:

```
{master}[edit]
jnpr@R1-RE0# show class-of-service host-outbound-traffic
forwarding-class nc;
dscp-code-point 111000;
```

It's not a good idea to mix things like Telnet and ping along with OSPF hellos. It's best that the `host-outbound-traffic` option is not used.

Given this is rather heavy handed, and results in noncritical traffic such as FTP, pings, and Telnet sessions now competing with network control, the preferred way to alter outbound RE queue selection is with a firewall filter, as described in Chapter 3. For example, this filter places all TCP-based control traffic, initial or retransmissions, into the NC queue and keeps the rest of the default behavior unmodified:

```
{master}[edit]
jnpr@R1-RE0# show interfaces lo0
unit 0 {
    family inet {
        filter {
            output RE_CLASSIFIER_OUT;
    }
}

{master}[edit]
jnpr@R1-RE0# show firewall family inet filter RE_CLASSIFIER_OUT
term BGP {
    from {
        protocol tcp;
        port bgp;
    }
    then {
        forwarding-class nc;
        accept;
    }
}
term LDP {
    from {
        protocol tcp;
        port ldp;
    }
    then {
        forwarding-class nc;
        accept;
    }
}
term everything-else {
    then accept;
}
```

Dynamic Profile Overview

MX platforms are often used to support subscriber access networks. Often, some form of DSLAM is used, along with RADIUS, to provide user access and authentication into the network. Operators can choose to provide static H-CoS to these dynamic users, perhaps leveraging the power of the remaining profile construct to provide basic CoS for dynamic users. Alternatively, dynamic CoS can be provided as a function of authentication based on RADIUS Vendor Specific Attributes (VSAs), which are used to map the user's authentication results to a set of CoS parameters.

A dynamic profile is a set of characteristics, defined in a type of template that's used to provide dynamic subscriber access and services for broadband applications. These services are assigned dynamically to interfaces. The `dynamic-profiles` hierarchy appears at the top level of the CLI hierarchy and contains many Junos configuration statements that you would normally define statically.

Dynamic profile statements appear in the following subhierarchies within the [edit dynamic-profiles] hierarchy. In the v11.4R1 release, these options are available:

```
{master}[edit]
jnpr@R1-RE0# set dynamic-profiles test ?
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
> class-of-service     Class-of-service configuration
> firewall             Define a firewall configuration
> interfaces           Interface configuration
> policy-options       Routing policy option configuration
> predefined-variable-defaults  Assign default values to predefined variables
> profile-variable-set  Dynamic profiles variable configuration
> protocols            Routing protocol configuration
> routing-instances    Routing instance configuration
> routing-options      Protocol-independent routing option configuration
> variables            Dynamic variable configuration
  |                    Pipe through a command
```

There are many options that allow you to customize a user's experience; here, the focus is on CoS-related operation, but generally speaking you will combine CoS with other dynamic profile functions to completely flesh out a service definition.

Dynamic Profile Linking. You can identify subscribers statically or dynamically. To identify subscribers statically, you can reference a static VLAN interface in a dynamic profile. To identify subscribers dynamically, you create variables for demultiplexing (demux) interfaces that are dynamically created when subscribers log in.

A demux interface can be statically or dynamically created. The demux interface is a logical interface that shares a common, underlying logical interface (in the case of IP demux) or underlying physical interface (in the case of VLAN demux). You can use these interfaces to identify specific subscribers or to separate individual circuits by IP address (IP demux) or VLAN ID (VLAN demux).

Dynamic CoS

Once authenticated to a basic CoS profile, subscribers can use RADIUS change-of-authorization (CoA) messages to activate a subscriber-specific service profile that includes customized values for key CoS parameters such as:

Shaping rate
Delay buffer rate
Guaranteed rate
Scheduler map

Optionally, you can configure default values for each parameter. Configuring default values is beneficial if you do not configure RADIUS to enable service changes. During service changes, RADIUS takes precedence over the default value that is configured.

Generally speaking, to deploy dynamic CoS you must first manually configure your network infrastructure for basic CoS, which includes all the classification, rewrites, scheduler settings, scheduler maps, IFL sets, etc., as detailed in this chapter. Once you have a CoS-enabled network that works to your design specifications, you then build on top of this infrastructure by adding dynamic CoS profiles. A detailed discussion of dynamic CoS is outside the scope of this chapter. Information on dynamic H-CoS in the v11.4 Junos release is available at the following URLs:

> *http://www.juniper.net/techpubs/en_US/junos11.4/topics/task/configuration/cos-subscriber-access-dynamic-summary.html*
> *http://www.juniper.net/techpubs/en_US/junos11.4/topics/concept/cos-subscriber-access-guidelines.html*

H-CoS Summary

The H-CoS architecture, as supported on fine-grained queuing Trio MPCs, is a powerful feature designed to provide a flexible and scalable CoS solution in B-RAS subscriber access applications where triple-play or business class offerings are enabled through IP CoS. The IFL-Set level of hierarchy is the heart and soul of H-CoS, as this new scheduling level allows you to apply CoS profiles to groupings of IFLs, thereby allowing you to lump subscribers into aggregate classes with specifically tailored guaranteed and peak rate parameters that map to services classes, and ultimately how much can be charged for the differentiated service levels. H-CoS at the IFL-Set level is a perfect way to offer the so-called olympic levels of service, namely gold, silver, and bronze.

H-CoS supports the notion of remaining CoS sets, with support for IFL-Sets, IFLs, and queues. Given that all MPC types have a finite number of queues and level 1/2/3 scheduler nodes, the ability to share these resources through a remaining group of IFL-Sets, IFLs, or queues, that would otherwise have no explicit CoS configuration, is a critical component in the ability to scale CoS to tens of thousands of subscriber ports.

H-CoS alone is pretty awesome. When you add support for dynamic CoS through RADIUS VSAs, a world of opportunities open up. For example, you can provide a “turbo charge” feature that allows a user to boost the parameters in their CoS profile through a resulting authentication exchange, in effect giving them a taste of the performance available at the higher tiers, which is an effort toward upselling the service tier or possibly adding per-use charging premiums for turbo mode, once the user is addicted to VoD or other high-capacity services. And all this is possible at amazing scale with MX Trio technology.

Trio Scheduling and Queuing

The scheduling stage determines when a given queue is serviced, in which order, and how much traffic can be drained at each servicing. In the Trio architecture, schedulers and queues are no longer closely linked, in that you can now have schedulers and shapers at all four levels of the H-CoS hierarchy, with queues only found at level 4.

When you configure a scheduler, you can define parameters for each of up to eight queues. These parameters include the scheduling priority, maximum queue depth/temporal delay, transmit rate, a peak rate (shaping), and how (or if) excess bandwidth is shared. At the queue level, you can also link to one or more WRED profiles; only queue-level schedulers support WRED profile linking. With H-CoS you can also define scheduling and shaping parameters at other levels of the hierarchy, though the specific parameters supported can vary by the node's position in the hierarchy.

Scheduling Discipline

MX routers with Trio-based MPCs use a form of Priority Queue Deficit Weighted Round Robin (PQ-DWRR) scheduling with five levels of strict priority. PQ-DWRR extends the basic deficit round robin (DWRR) mechanism by adding support for one or more priority queues that exhibit minimal delay. The deficit part of the algorithm's name stems from the allowance of a small amount of negative credit in an attempt to keep queues empty. The resultant negative balance from one servicing interval is carried over to the next quantum's credit allocation, keeping the average dequeueing rate near the configured transmit value.

Strict priority means that a configured transmit rate, or weight, is only relevant among queues at the same priority level. Within a given priority, weighted round robin is performed, but the next lower priority level is only serviced when all queues at the current and higher priority level are empty (or have met their shaping limits). With this type of scheduler, starvation is always a potential if using more than one priority without some form of rate limit or shaping in effect. In Trio, queues that have reached their transmit rate automatically drop their priorities, a mechanism that helps prevent high priority queues from starving out lower ones.

APQ-DWRR scheduler is defined by four variables:

Buffer size

This is the delay buffer for the queue that allows it to accommodate traffic bursts. You can configure a buffer size as a percentage of the output interface's total buffer capacity, or as a temporal value from 1 to 200,000 microseconds, which simply represents buffer size as a function of delay, rather than bytes. The value configured is mapped into the closest matching hardware capability. Most Trio PFEs offer 500 milliseconds of delay bandwidth buffer, based on a 4 × 10GE MIC; each port is preassigned 100 milliseconds of that buffer with the balance left for assignment via the CLI.

The quantum

The quantum is the number of credits added to a queue every unit of time and is a function of the queue transmit rate. The actual quantum used by Trio varies by hardware type and the level of the scheduling node in the H-CoS hierarchy; in general it's based on a 21 millisecond interval. The queue's transmit rate specifies the amount of bandwidth allocated to the queue and can be set based on bits per second or as a percentage of interface bandwidth. By default, a queue can be serviced when in negative credit, as long as no other queues have traffic pending and it's not blocked from excess usage. When desired, you can shape a queue to its configured transmit rate with inclusion of the `exact` keyword, or rate limit the queue using a policer via the `rate-limit` option.

Priority

The priority determines the order in which queues are serviced. A strict priority scheduler services high-priority queues in positive credit before moving to the next level of priority. In Trio, in-profile queues of the same priority are serviced in a simple round-robin manner, which is to say one packet is removed from each queue while they remain positive. This is opposed to WRR, which would dequeue *n* packets per servicing with *n* being based on the configured weight. When a queue exceeds its transmit rate, it can send at an excess low or high priority level based on configuration. Excess bandwidth sharing on Trio is based on a weighting factor, which means that WRR scheduling is used to control excess bandwidth sharing on trio.

A `strict-high` priority queue is a special case of high priority, where the effective transmit weight is set to equal egress interface capacity. This means that a strict-high queue can never go negative and therefore is serviced before any low-priority queue anytime it has traffic waiting. The result is known as low-latency queuing (LLQ). Care should be used when a queue is set to strict high to ensure that the queue does not starve low-priority traffic; a strict high queue should be limited using either the `exact` keyword to shape it, the `rate-limit` keyword to police it, or some external mechanism such as a policer called through a filter to ensure starvation does not occur.

In the v11.4 release, only one queue can be designated as strict-high in a given scheduler map, otherwise the following commit error is generated:

```
[edit]
jnpr@R4# commit
[edit class-of-service]
  'scheduler-maps sched_map_core'
    More than one schedulers with priority strict-high for scheduler-map
sched_map_core
error: configuration check-out failed
```

When you have two or more queues set to high priority (both at high, or one high and one strict high), the PQ-DWRR scheduler simply round-robins between them until one goes negative due to having met its transmit rate (in the case of high) or for strict high when the queue is empty or has met its rate limit. When the remaining high-priority queue is serviced, the scheduler can move on to the next scheduling priority level.

> Shaping adds latency/delays and is therefore not the ideal way to limit traffic in a LLQ. Consider Connection Admission Control (CAC) mechanisms such as ingress policing or hard rate limiting.

Deficit counter

PQ-DWRR uses the deficit counter to determine whether a queue has enough credits to transmit a packet. It is initialized to the queue's quantum, which is a function of its transmit rate, and is the number of credits that are added to the queue every quantum.

Scheduler Priority Levels

As noted previously, Trio PFEs support priority-based MDWRR at five priority levels. These are:

Guaranteed high (GH)
Guaranteed medium (GM)
Guaranteed low (GL)
Excess high (EH)
Excess low (EL)

The first three levels are used for traffic in the guaranteed region, which is to say traffic that is sent within the queue's configured transmit rate. Once a queue has reached the configured guaranteed rate for a given guaranteed level, it either stops sending, for example in the case of a hard rate limit using `exact`, or it transitions to one of the two excess regions based on the queue's configuration. When so desired, you can configure a queue with 0 transmit weight such that it may *only* send excess traffic, or if desired prevent a queue from using any excess bandwidth using either the `exact`, `rate-limit`, or `excess-priority none` options. Figure 5-16 shows how priority-based scheduling works over the different priorities.

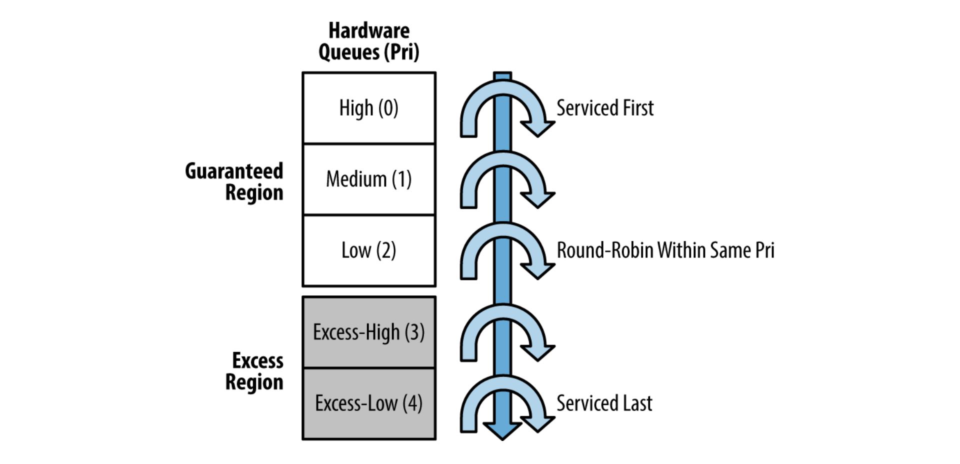


Figure 5-16. Trio Priority-Based Scheduling.

The figure shows the five scheduler priorities that are found at all levels of the Trio scheduling hierarchy. While only one set is shown, recall that scheduling can occur in three places in the current H-CoS architecture. However, due to priority inheritance (described in the following), it can be said that at any given time all three levels select the same priority level, which is based on the queue with the highest priority traffic pending, such that we can focus on a single layer for this discussion. It's important to note that queues and hardware priority are a many-to-one mapping, in that a single priority level can be assigned to all queues if desired; note that such a setting effectively removes the PQ from the PQ-DRR scheduling algorithm as it places all queues on an equal footing.

The scheduler always operates at the highest priority level that has traffic pending. In other words, the scheduler first tries to provide all high-priority queues with their configured transmit weights before moving down to the next priority level where the process repeats. Priority levels that have no active queues are skipped in a work-conserving manner. Traffic at the lowest priority level (excess-low) can only be sent when the sum of all traffic for all active priorities above it is less than the interface's shaping rate. As previously noted, priority-based scheduling can lead to starvation of lower classes if steps are not taken to limit the capacity of higher priority queues.

Once a given priority level becomes active, the scheduler round-robins between all the queues at that level, until a higher level again interrupts to indicate it has new traffic pending or until all queues at the current priority level are empty, the latter allowing the scheduler to move on to service traffic at the next lowest priority.

Scheduler to Hardware Priority Mapping

While Trio hardware supports five distinct hardware priorities, the Junos CLI is a bit more flexible in that it predates Trio and supports a wide range of networking devices.

As such, not all supported Junos CLI scheduler or TCP priorities map to a corresponding hardware scheduling priority. Table 5-11 shows the mappings between Trio and Junos CLI priorities.

Table 5-11. Scheduler to Hardware Priority Mappings.

CLI Scheduler Priority	HW Pri: TX < CIR	HW Pri: TX > CIR	Comment
Strict-high	0	0	GH: Strict-high and high have same hardware priority, but strict-high is not demoted as it should be shaped or rate limited to transmit rate.
High	0	3/4	GH: Excess level depends on configuration, default is GH.
Medium-high Medium-low	1	3/4	GM: Excess level depends on configuration, default is EL. The two medium CLI priorities share a HW priority. Can use RED/PLP to differentiate between the two.
Low	2	3/4	GL: Excess level depends on configuration. Cannot combine with excess none.
Excess priority high	NA	3	EH: Trio has two excess levels; this is the highest of them.
Excess priority medium-high	NA	3	NA: Commit error, not supported on Trio.
Excess priority medium-low	NA	4	NA: Commit error, not supported on Trio.
Excess priority low	NA	4	GL: The lowest of all priorities.
Excess priority none	NA	NA, excess traffic is not permitted.	Traffic above CIR is queued (buffered), can be sent only as G-Rate.

In Table 5-11, hardware priority levels 0 to 2 reflect in-profile or guaranteed rate traffic, which is to say traffic at or below the queue's transmit rate.

Guaranteed High (GH), or just high, is the highest of the hardware priorities and is shared by both high and strict high. The difference is that by default Strict High (SH) gets 100% of an interface's bandwidth if not rate limited or shaped, and therefore it cannot exceed transmit rate and go negative at the queue level. A SH queue that is shaped or rate limited is also unable to exceed its limits, again not going into excess at the queue level.

Testing shows that both SH and H are subjected to priority demotion at scheduler nodes when per priority shaping rates are exceeded.

The `medium-high` and `medium-low` scheduler priorities both map to Guaranteed Medium (GM) and so share a hardware priority level, which affords them the scheduling behavior from a priority perspective. You can set medium-low to have a high PLP and then use a WRED profile to aggressively drop PLP high to differentiate between the two medium levels of service if desired.

Priorities 3 and 4 represent the Excess High (EH) and Excess Low (EL) priorities. This level is used by queues for traffic above the queue's transmit rate, but below the queue's shaping or PIR rate. Whenever the queue is sending above its transmit rate, it switches to either EH or EL, based on configuration. EH and EL can also represented any guaranteed level priority (GH, GM, GL), after it has been demoted at a scheduler node in response to exceeding per priority shapers.

You can block both behaviors with an excess setting of none, which is detailed later.

Priority Propagation

As mentioned previously, a queue demotes its own traffic to the configured excess region once it has met the configured transmit rate. A queue that is set for strict high is the exception. Such a queue is provided with a transmit rate that equals the interface's rate (shaped or physical), and as such cannot go negative. Typically, a strict high queue is rate limited or shaped (to prevent starvation of lesser priority queues), but any queue that is rate limited/shaped to its transmit rate must also remain within the guaranteed region and is therefore inherently limited from excess bandwidth usage.

Therefore, in H-CoS, the priority of a queue is determined strictly by its scheduler configuration (the default priority is low), and whether the queue is within its configured transmit rate. However, the other hierarchical schedulers do not have priorities explicitly configured. Instead, the root and internal scheduling nodes inherit the priority of their highest priority grandchild/child through a mechanism called priority propagation. The inherited priority may be promoted or demoted at Level 3 nodes, in a process described subsequently.

The specific priority of a given node is determined by:

The highest priority of an active child.
If an L3 node, also factor whether above its configured guaranteed rate (CIR) (only relevant if the physical interface is in CIR mode).

The result is that a high-priority queue should only have to wait for a period of time that is equal to the transmission time of a single MTU-sized packet before the L1, L2, and L3 schedulers sense a higher priority child is active via priority propagation. As soon as the current low-priority frame is transmitted, the scheduler immediately jumps back to the highest active priority level where it can service the new high-priority traffic. Figure 5-17 shows the priority propagation process.

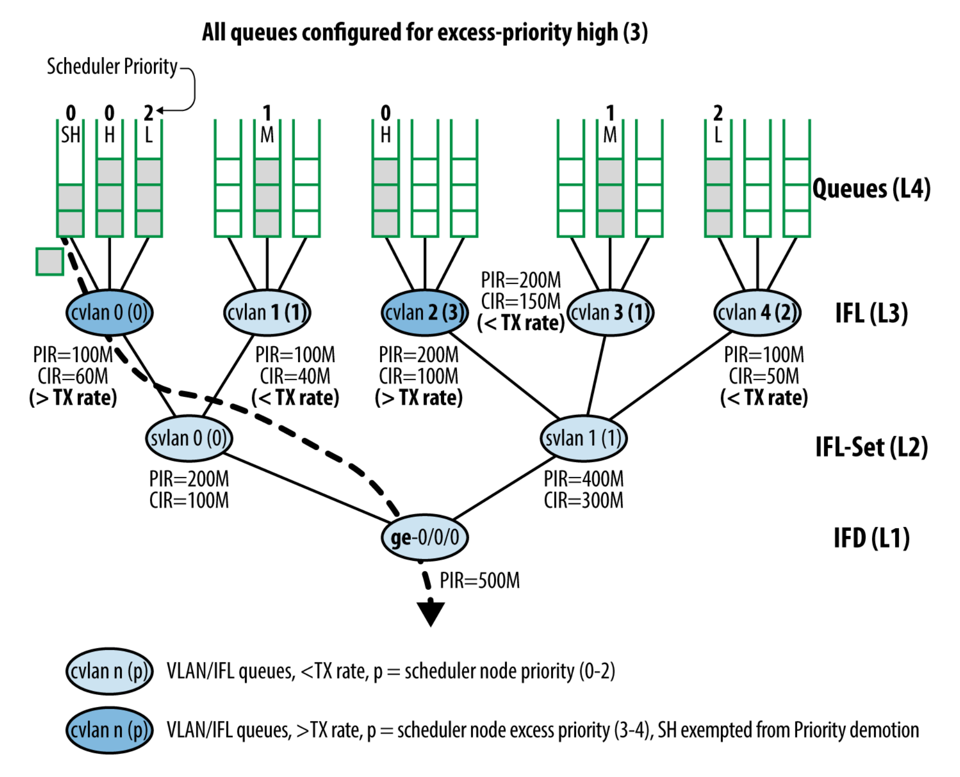


Figure 5-17. Scheduler Node Priority Propagation Part 1.

The figure holds a wealth of information, so let's begin with an overview. First, note that a full four-level hierarchical scheduler model is shown and that it's based on S-VLAN IFL-Sets at level 2 and C-VLAN based IFLs at level 3, each with a set of queues at level 4. The queues have a mix of SH, H, M, and L priority schedulers configured. The queues with traffic pending have their configured priority shown, both in CLI terms of H, M, and L, and using the Trio hardware priority values that range from 0 to 2 for traffic below the transmit rate. All queues are configured for an excess priority of high, with the exception of the SH queue as it can technically never enter an excess region, making such a setting nonapplicable.

The scheduler nodes at level 2 and 3 are shown with both a PIR (shaped rate) as well as a CIR (guaranteed rate). In keeping with best practice, note that the sum of the CIRs assigned to a node's children is then carried down to become the lower level node's CIR. This is not necessarily the case with the PIRs, which may or may not be set to equal or excess the sum of the node's child PIRs. For example, note how the sum of PIRs for the two S-VLAN nodes sums to 600 Mbps, which is 100 Mbps over the PIR configured at the IFD level. While the two IFL-Sets can never both send at their shaped rates simultaneously, the sum of their CIRs is below the IFD's shaped rate, and no CIRs are overbooked in this example, thereby making the guaranteed rates achievable by all queues simultaneously. Such a practice makes good CoS sense; given that PIR is a maximum and not a guaranteed rate, many users do not feel compelled to ensure that all nodes can send at their shaped rates at the same time. Generally, the opposite is true for CIR or guaranteed rates. In order to ensure that all nodes can obtain their guaranteed rate at the same time, you must ensure that the CIR of one node is not overbooked by a lower level node.

Junos H-CoS allows you to overbook CIR/guaranteed rates when in PIR/CIR mode. While supported, overbooking of CIR is not recommended. Overbooked CIR is not supported in per unit scheduling modes and is not applicable to port-level CoS, which has only PIR mode support.

> While you can overbook a queue's shaping rate (PIR) with respect to its level 3 (IFL) node's PIR, it should be noted that Junos is expected to throw a commit error if the sum of the queue transmit rates (as a percentage or absolute) exceeds a lower level node's PIR, as such a configuration guarantees that the queue's transmit rates can never be honored. In testing the v11.4R1 release, it was found that the expected commit error was not seen unless a Remaining Traffic Profile (RTP) with a `scheduler-map` is used. PR 784970 was opened to restore the expected commit error when the queue transmit rate exceeds either IFL or IFL-Set shaped rates.

The instant of time captured in the figure has the queues for C-VLANs 0 and 2 when they have just met their configured transmit weight. In this example, the SH queue does not have a rate limit, making its transmit rate statement meaningless as it inherits the full 100% rate of the interface, but we still speak in terms of it being above or below its transmit weight to help explain the resulting behavior.

The scheduler node for C-VLAN 0 senses the SH queue has traffic pending, which means that of all queues in the hierarchy it's got the highest current priority (again, despite it being above the transmit rate that is set due to being SH). Inheritance results in the queue's SH priority (0) value being inherited by the S-VLAN 0 scheduler node at level 2, a process repeated at the IFD level scheduler, resulting in the SH queue being serviced first, as indicated by the dotted line.

Meanwhile, over at C-VLAN 2, the high-priority queue exceeded its transmit rate leading to demotion into excess priority high, causing its level 3 scheduler node to inherit that same value, as it's currently the highest priority asserted by any of its children (the queues).

The sequence continues in Figure 5-18.

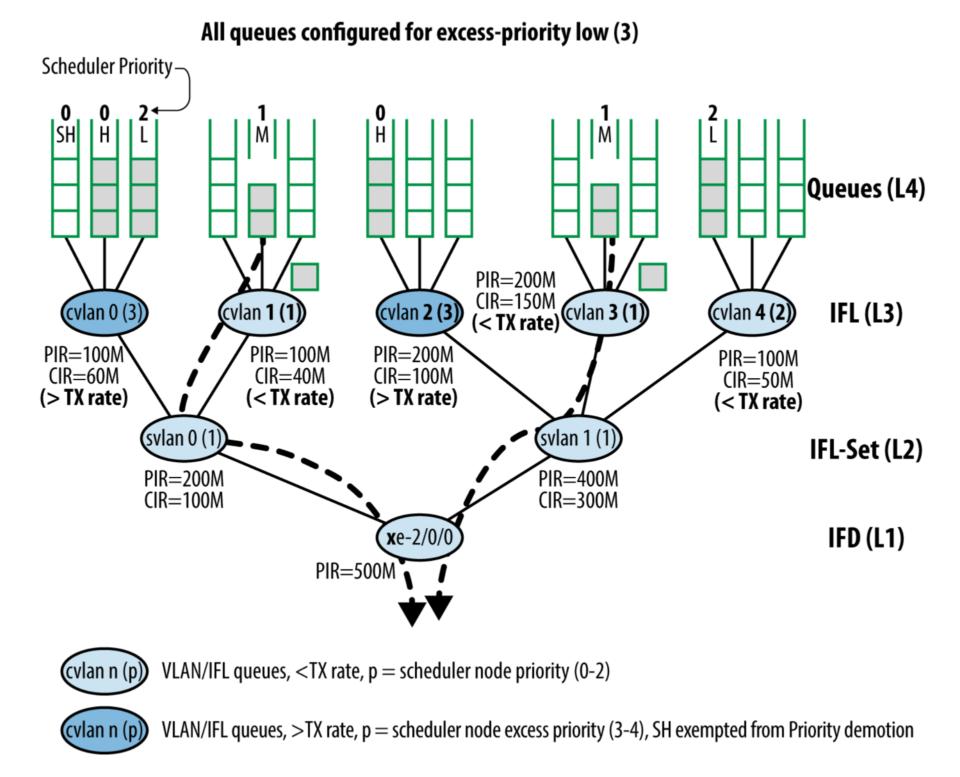


Figure 5-18. Scheduler Node Priority Propagation Part 2.

Because C-VLAN 3 has not entered into excess range yet, it remains at M priority (1). As a result, the L2 scheduler node for S-VLAN 1 inherits the highest active priority, which here is the medium-priority grandchild (queue) at C-VLAN 3, resulting in that queue being serviced, as represented by the second dotted line.

In this stage of events, the SH queue at C-VLAN 0 is empty. Given all other queues have met their transmit rates at this node, the priority drops to the configured excess low range, causing its level 3 scheduler node to inherit the priority value 3. With no level 0 priority queues pending, the scheduler drops to the next level and services C-VLANs 1 and 3; as these queues are at the same priority, the scheduler will round-robin between them, dequeueing one packet on each visit until the queues are either empty or reach their transmit rate, causing their priority to drop.

Figure 5-19 completes the sequence.

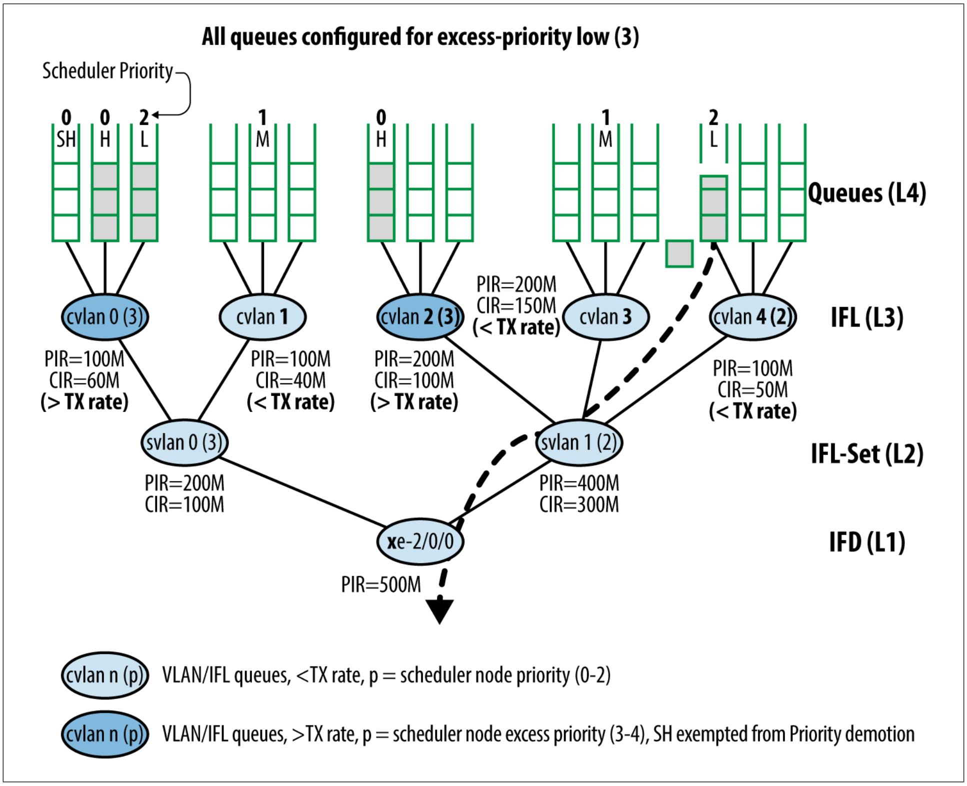


Figure 5-19. Scheduler Node Priority Propagation Part 3.

In Figure 5-19, all H and M priority queues have been emptied. This makes C-VLAN 4's low-priority queue, at priority 2, the highest active priority in its level 3 and level 2 parent/grandparent nodes, resulting in the low-priority queue being serviced.

Though not shown, with all in-profile queues serviced and SH still empty, the scheduler moves on to service excess traffic at C-VLANs 0 and 2. Because all queues have been set to use the same excess priority in this example, the result is the scheduler moves on to service the H/L queues at C-VLAN 0 and the H queue at C-VLAN 2 in round-robin fashion according to their excess weights, rather than simple round-robin. If during this time another queue asserts a higher priority, the inheritance mechanism ensures that queue is serviced as soon as the current packet is dequeued, thus ensuring only one MTU worth of serialization delay for higher priority traffic at this scheduling block.

Priority Promotion and Demotion. Level 2 and Level 3 scheduler nodes have the ability to promote excess levels into the guaranteed low region, as well as to demote guaranteed low into an excess region, as described previously in the "The H-CoS Reference Model" on page 350 .

Scheduler nodes perform both types of demotion: G-Rate and per priority shaping. The former occurs when a scheduling node's G-Rate credits fall low and it demotes GL into the excess region. The second form can occur for GH, GM, or GL, as a function of a per priority shaper, which demotes traffic in excess of the shaper into an excess region. Promotion and demotion of G-Rate at L1 scheduler nodes is not supported, as these nodes have no concept of a guaranteed rate in the current H-CoS reference model.

At the queue level, priority is handled a bit differently. Queues don't factor scheduler node G-Rates; instead, they perform promotion/demotion based strictly on whether they are sending below or above their configured transmit rate. Queue-level priority demotion occurs for GH, GM, and GL, with the specific excess priority level being determined by a default mapping of SH/H to EH and M/L to EL, or by explicit configuration at the queue level.

Queues versus Scheduler Nodes

It's easy to gets queues and scheduler nodes confused, but they are distinctly different. Queues are that which is scheduled and which hold the actually notification cells that represent packets slated for transmission. Scheduler nodes, in contrast, exist to select which queue is to be serviced based on its relative priority, a function that varies depending on if the queue is within, or above, its configured transmit rate.

Another key difference is the way priority is handled. Priority is explicitly configured at queues only; scheduler nodes inherit the priority of the queue being serviced. Priority demotion is handled differently at queues versus scheduler nodes, as described in this section, and only a scheduler node can promote priority based on exceeding a G-Rate or per priority shaper.

Scheduler Modes

A Trio interface can operate in one of three different scheduling modes. These are port-level, per unit, and hierarchical modes. All MPC types support per port scheduling, but only the dense queuing MPCs are capable of the per unit or hierarchical scheduling modes.

Port-Level Queuing

In per port scheduling mode, a single scheduler node at the IFD level (level 1) services a set of queues that contain traffic from all users via a single IFL. This is illustrated in Figure 5-20.

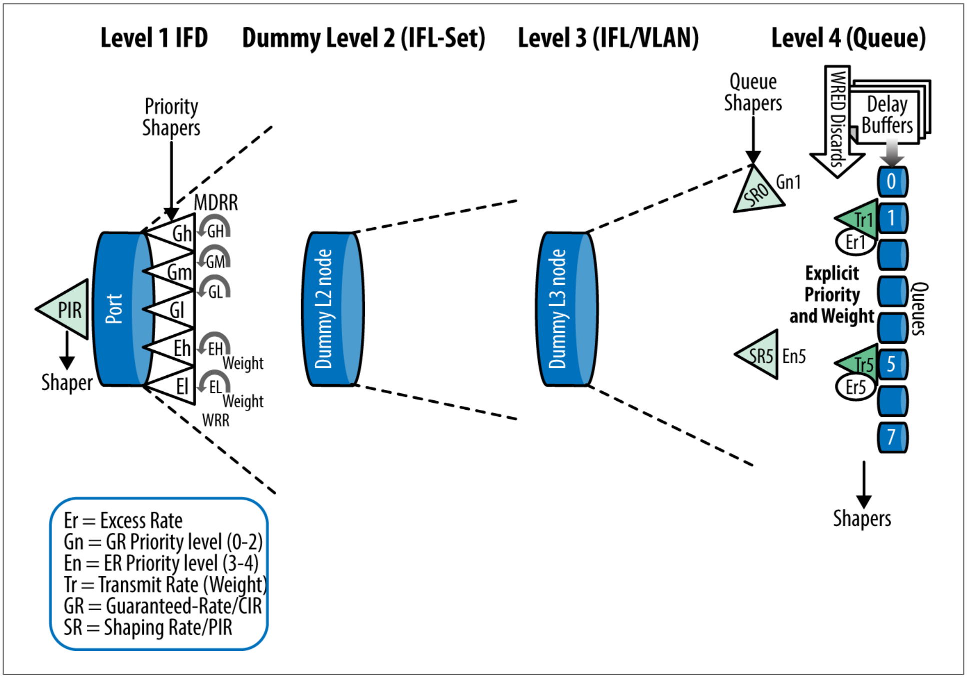


Figure 5-20. Trio Port-Based Queuing.

Port-level scheduling is the default mode of operation when you do not include a `per-unit-scheduler` or `hierarchical-scheduler` statement at the `[edit interfaces <name>]` hierarchy. As previously noted, all Trio MPCs are capable of operating in port-level scheduling mode, and while the overall operation is the same, there are slight differences in the way things are implemented on queuing versus nonqueuing cards. On the former, the Dense Queuing ASIC (QX) handles the scheduling, while in the latter the function is pushed off to the Buffer Management ASIC (MQ) as there is no queuing ASIC present. When performed on a queuing MPC, port-level scheduling incorporates a dummy level 2 and level 3 scheduling node, as shown in Figure 5-20.

> Because PIR mode requires a shaper with a guaranteed rate be attached at either L2 or L3, and in port mode these are dummy nodes that do not accept a TCP, port mode operation is always said to operate in PIR mode. Currently, L1 nodes (IFD) can only shape to a PIR using a TCP, and so the L1 scheduler does not support CIR mode.

Port-level CoS supports queue shaping, as well as a IFD and per priority shapers at the L1 node. An example port-level CoS configuration is shown:

```
{master}[edit]
jnpr@R1-RE0# show class-of-service schedulers
```

```
sched_ef_50 {
    transmit-rate 2m rate-limit;
    buffer-size temporal 25k;
    priority strict-high;
}
sched_af4x_40 {
    transmit-rate 1m;
    excess-rate percent 40;
    excess-priority high;
}
sched_af3x_30 {
    transmit-rate 1m;
    excess-rate percent 30;
    excess-priority low;
}
sched_af2x_10 {
    transmit-rate 1m;
    excess-rate percent 10;
    excess-priority low;
}
sched_af1x_5 {
    transmit-rate 1m;
    excess-rate percent 5;
    excess-priority low;
}
sched_be_5 {
    transmit-rate 1m;
    shaping-rate 3m;
    excess-priority low;
}
sched_nc {
    transmit-rate 500k;
    buffer-size percent 10;
    priority high;
}

{master}[edit]
jnpr@R1-RE0# show class-of-service scheduler-maps
sched_map_pe-p {
    forwarding-class ef scheduler sched_ef_50;
    forwarding-class af4x scheduler sched_af4x_40;
    forwarding-class af3x scheduler sched_af3x_30;
    forwarding-class af2x scheduler sched_af2x_10;
    forwarding-class af1x scheduler sched_af1x_0;
    forwarding-class be scheduler sched_be_5;
    forwarding-class nc scheduler sched_nc;

jnpr@R1-RE0# show class-of-service interfaces
xe-2/0/0 {
    output-traffic-control-profile TCP_PE-P_5;
    unit 0 {
        classifiers {
            ieee-802.1 ieee_classify;
        }
        rewrite-rules {
            ieee-802.1 ieee_rewrite;
        }
    }
    unit 1 {
        classifiers {
            dscp dscp_diffserv;
        }
        rewrite-rules {
            dscp dscp_diffserv;
        }
    }
}
. . .
{master}[edit]
jnpr@R1-RE0# show interfaces xe-2/0/0
vlan-tagging;
unit 0 {
    family bridge {
        interface-mode trunk;
        vlan-id-list 1-999;
    }
}
unit 1 {
    vlan-id 1000;
    family inet {
        address 10.8.0.0/31;
    }
    family iso;
}
```

Note how the IFD, xe-2/0/0 in this example, has no `per-unit-scheduler` or `hierarchical-scheduler` statements, placing it into the default port-level mode. The interface is using a mix of Layer 2 and Layer 3 rewrite and classification rules, applied to unit 0 and 1, respectively, given the trunk between R1 and R2 supports both bridged and routed traffic. A pure Layer 2 port, such as the S1-facing xe-2/2/0, needs only the L2 rules; in like fashion, a Layer 3 interface such as xe-2/1/1 requires only the L3 classification and rewrite rules.

A single Traffic Control Profile (TCP) is applied at the IFD level; no unit number is specified for the interface, and attempting to do so generates a commit error unless per unit or hierarchical mode is in effect. As a result, both the L2 and L3 unit on the IFD share the 5 Mbps shaper and a single set of queues. Traffic from all seven FCs/queues is scheduled into the shaped bandwidth based on each queues transmit rate, scheduled according to the queue's priority, with excess bandwidth shared according to the specified percentages.

Here the EF queue is rate limited (not shaped) to ensure it cannot starve lesser classes. It's also provisioned with a buffer based on temporal delay, in microseconds, again to help limit per node delay and therefore place a cap on end-to-end delays for the real-time service. The network control (NC) queue has been granted a larger delay buffer than its 500 kbps transmit rate would normally have provided to help ensure less a greater chance of delivery during congestion, albeit at the cost of a potentially higher queuing delay; given that network control is not real-time, additional delay in favor of reduced loss is generally a good trade.

In this port mode CoS example, the transmit rates are set to an absolute bit rate rather than a percentage. In such cases, the sum of queue bandwidth is allowed to exceed the L1 node's shaping rate, such as in this case where the total queue bandwidth sums to 7.5 Mbps while the IFD is shaped to 5 Mbps.

This is not the case when rates are specified as a percentage, where the sum cannot exceed 100%. For example, here the rates are converted to percentages that exceed 100% and a commit error is returned:

```
sched_ef_50 {
    transmit-rate percent 50 rate-limit;
    buffer-size temporal 25k;
    priority strict-high;
}
sched_af4x_40 {
    transmit-rate percent 20;
    excess-rate percent 40;
     excess-priority high;

}
sched_af3x_30 {
    transmit-rate percent 20;
    excess-rate percent 30;
     excess-priority low;

}
sched_af2x_10 {
    transmit-rate percent 20;
    excess-rate percent 10;
     excess-priority low;

}
sched_af1x_5 {
    transmit-rate percent 20;
    excess-rate percent 5;
    buffer-size percent 10;
     excess-priority low;
}
sched_be_5 {
    transmit-rate percent 5;
    shaping-rate 3m;
    buffer-size percent 5;
    excess-priority low;
}
sched_nc {
    transmit-rate percent 5;
    priority high;
    buffer-size percent 10;
}
```

```
{master}[edit class-of-service scheduler-maps sched_map_pe-p]
jnpr@R1-RE0# commit
re0:
[edit class-of-service interfaces]
  'xe-2/0/0'
    Total bandwidth allocation exceeds 100 percent for scheduler-map sched_map_pe-p
error: configuration check-out failed
```

Operation Verification: Port Level. Standard CLI show commands are issued to confirm the TCP is applied to the IFD, and to verify the L2 and L3 rewrite and classification rules:

```
{master}[edit]
jnpr@R1-RE0# run show class-of-service interface xe-2/0/0
Physical interface: xe-2/0/0, Index: 148
Queues supported: 8, Queues in use: 8
  Output traffic control profile: TCP_PE-P_5, Index: 28175
  Congestion-notification: Disabled

  Logical interface: xe-2/0/0.0, Index: 332
    Object                  Name                   Type                   Index
    Rewrite                 ieee_rewrite            ieee8021p (outer)      16962
    Classifier              ieee_classify          ieee8021p              22868

  Logical interface: xe-2/0/0.1, Index: 333
    Object                  Name                   Type                   Index
    Rewrite                 dscp_diffserv          dscp                   23080
    Classifier              dscp_diffserv          dscp                   23080

  Logical interface: xe-2/0/0.32767, Index: 334
```

The display confirms the classifiers and rewrite rules, and the application of a TCP at the IFD level. The TCP and `scheduler-map` is displayed to confirm the IFD shaping rate, as well as queue priority and bandwidth, delay buffer, and WRED settings:

```
{master}[edit]
jnpr@R1-RE0# run show class-of-service traffic-control-profile TCP_PE-P_5
Traffic control profile: TCP_PE-P_5, Index: 28175
  Shaping rate: 5000000
  Scheduler map: sched_map_pe-p

{master}[edit]
jnpr@R1-RE0# run show class-of-service scheduler-map sched_map_pe-p
Scheduler map: sched_map_pe-p, Index: 60689

  Scheduler: sched_be_5, Forwarding class: be, Index: 4674
    Transmit rate: 1000000 bps, Rate Limit: none, Buffer size: remainder,
      Buffer Limit: none, Priority: low
    Excess Priority: low
    Shaping rate: 3000000 bps
    Drop profiles:
      Loss priority   Protocol    Index    Name
      Low             any             1    <default-drop-profile>
      Medium low      any             1    <default-drop-profile>
      Medium high     any             1    <default-drop-profile>
      High            any             1    <default-drop-profile>
```

```
Scheduler: sched_af1x_5, Forwarding class: af1x, Index: 12698
  Transmit rate: 1000000 bps, Rate Limit: none, Buffer size: remainder,
    Buffer Limit: none, Priority: low
  Excess Priority: low, Excess rate: 5 percent,
  Drop profiles:
    Loss priority   Protocol    Index    Name
    Low             any             1    <default-drop-profile>
    Medium low      any             1    <default-drop-profile>
    Medium high     any             1    <default-drop-profile>
    High            any             1    <default-drop-profile>

Scheduler: sched_af2x_10, Forwarding class: af2x, Index: 13254
  Transmit rate: 1000000 bps, Rate Limit: none, Buffer size: remainder,
    Buffer Limit: none, Priority: low
  Excess Priority: low, Excess rate: 10 percent,
  Drop profiles:
    Loss priority   Protocol    Index    Name
    Low             any             1    <default-drop-profile>
    Medium low      any             1    <default-drop-profile>
    Medium high     any             1    <default-drop-profile>
    High            any             1    <default-drop-profile>

Scheduler: sched_nc, Forwarding class: nc, Index: 25664
  Transmit rate: 500000 bps, Rate Limit: none, Buffer size: 10 percent,
    Buffer Limit: none, Priority: high
  Excess Priority: unspecified
  Drop profiles:
    Loss priority   Protocol    Index    Name
    Low             any             1    <default-drop-profile>
    Medium low      any             1    <default-drop-profile>
    Medium high     any             1    <default-drop-profile>
    High            any             1    <default-drop-profile>

Scheduler: sched_af4x_40, Forwarding class: af4x, Index: 13062
  Transmit rate: 1000000 bps, Rate Limit: none, Buffer size: remainder,
    Buffer Limit: none, Priority: low
  Excess Priority: high, Excess rate: 40 percent,
  Drop profiles:
    Loss priority   Protocol    Index    Name
    Low             any             1    <default-drop-profile>
    Medium low      any             1    <default-drop-profile>
    Medium high     any             1    <default-drop-profile>
    High            any             1    <default-drop-profile>

Scheduler: sched_ef_50, Forwarding class: ef, Index: 51203
  Transmit rate: 2000000 bps, Rate Limit: rate-limit, Buffer size: 25000 us,
    Buffer Limit: exact, Priority: strict-high
  Excess Priority: unspecified
  Drop profiles:
    Loss priority   Protocol    Index    Name
    Low             any             1    <default-drop-profile>
    Medium low      any             1    <default-drop-profile>
    Medium high     any             1    <default-drop-profile>
    High            any             1    <default-drop-profile>
Scheduler: sched_af3x_30, Forwarding class: af3x, Index: 13206
  Transmit rate: 1000000 bps, Rate Limit: none, Buffer size: remainder,
    Buffer Limit: none, Priority: low
  Excess Priority: low, Excess rate: 30 percent,
  Drop profiles:
    Loss priority   Protocol    Index    Name
    Low             any             1    <default-drop-profile>
    Medium low      any             1    <default-drop-profile>
    Medium high     any             1    <default-drop-profile>
    High            any             1    <default-drop-profile>

Scheduler: sched_null, Forwarding class: null, Index: 21629
  Transmit rate: 0 bps, Rate Limit: none, Buffer size: remainder,
    Buffer Limit: none, Priority: medium-high
  Excess Priority: none
  Drop profiles:
    Loss priority   Protocol    Index    Name
    Low             any             1    <default-drop-profile>
    Medium low      any             1    <default-drop-profile>
    Medium high     any             1    <default-drop-profile>
    High            any             1    <default-drop-profile>
```

The show interfaces queue output confirms support for eight FCs and that currently only BE and NC are flowing:

```
{master}[edit]
jnpr@R1-RE0# run show interfaces queue xe-2/0/0
Physical interface: xe-2/0/0, Enabled, Physical link is Up
  Interface index: 148, SNMP ifIndex: 4373
Forwarding classes: 16 supported, 8 in use
Egress queues: 8 supported, 8 in use
Queue: 0, Forwarding classes: be
  Queued:
    Packets              :                 43648                     0 pps
    Bytes                :               5499610                     0 bps
  Transmitted:
    Packets              :                 43648                      0 pps
    Bytes                :               5499610                     0 bps
    Tail-dropped packets :                     0                     0 pps
    RED-dropped packets  :                     0                     0 pps
     Low                 :                     0                     0 pps
     Medium-low          :                     0                     0 pps
     Medium-high         :                     0                     0 pps
     High                :                     0                     0 pps
    RED-dropped bytes    :                     0                     0 bps
     Low                 :                     0                     0 bps
     Medium-low          :                     0                     0 bps
     Medium-high         :                     0                     0 bps
     High                :                     0                     0 bps
Queue: 1, Forwarding classes: af1x
  Queued:
    Packets              :                     0                     0 pps
    Bytes                :                     0                     0 bps
  Transmitted:
    Packets              :                     0                      0 pps
    Bytes                :                     0                     0 bps
    Tail-dropped packets :                    0                    0 pps
    RED-dropped packets  :                    0                    0 pps
     Low                 :                    0                    0 pps
     Medium-low          :                    0                    0 pps
     Medium-high         :                    0                    0 pps
     High                :                    0                    0 pps
    RED-dropped bytes    :                    0                    0 bps
     Low                 :                    0                    0 bps
     Medium-low          :                    0                    0 bps
     Medium-high         :                    0                    0 bps
     High                :                    0                    0 bps
Queue: 2, Forwarding classes: af2x
  Queued:
    Packets              :                    0                    0 pps
    Bytes                :                    0                    0 bps
  Transmitted:
    Packets              :                    0                     0 pps
    Bytes                :                    0                    0 bps
    Tail-dropped packets :                    0                    0 pps
    RED-dropped packets  :                    0                    0 pps
     Low                 :                    0                    0 pps
     Medium-low          :                    0                    0 pps
     Medium-high         :                    0                    0 pps
     High                :                    0                    0 pps
    RED-dropped bytes    :                    0                    0 bps
     Low                 :                    0                    0 bps
     Medium-low          :                    0                    0 bps
     Medium-high         :                    0                    0 bps
     High                :                    0                    0 bps
Queue: 3, Forwarding classes: nc
  Queued:
    Packets              :                 4212                   12 pps
    Bytes                :               396644                 9040 bps
  Transmitted:
    Packets              :                 4212                    12 pps
    Bytes                :               396644                 9040 bps
    Tail-dropped packets :                    0                    0 pps
    RED-dropped packets  :                    0                    0 pps
     Low                 :                    0                    0 pps
     Medium-low          :                    0                    0 pps
     Medium-high         :                    0                    0 pps
     High                :                    0                    0 pps
    RED-dropped bytes    :                    0                    0 bps
     Low                 :                    0                    0 bps
     Medium-low          :                    0                    0 bps
     Medium-high         :                    0                    0 bps
     High                :                    0                    0 bps
Queue: 4, Forwarding classes: af4x
  Queued:
    Packets              :                    0                    0 pps
    Bytes                :                    0                    0 bps
  Transmitted:
    Packets              :                    0                     0 pps
    Bytes                :                    0                    0 bps
    Tail-dropped packets :                    0                    0 pps
    RED-dropped packets  :                     0                     0 pps
     Low                 :                     0                     0 pps
     Medium-low          :                     0                     0 pps
     Medium-high         :                     0                     0 pps
     High                :                     0                     0 pps
    RED-dropped bytes    :                     0                     0 bps
     Low                 :                     0                     0 bps
     Medium-low          :                     0                     0 bps
     Medium-high         :                     0                     0 bps
     High                :                     0                     0 bps
Queue: 5, Forwarding classes: ef
  Queued:
    Packets              :                     0                     0 pps
    Bytes                :                     0                     0 bps
  Transmitted:
    Packets              :                     0                      0 pps
    Bytes                :                     0                     0 bps
    Tail-dropped packets :                     0                     0 pps
    RED-dropped packets  :                     0                     0 pps
     Low                 :                     0                     0 pps
     Medium-low          :                     0                     0 pps
     Medium-high         :                     0                     0 pps
     High                :                     0                     0 pps
    RED-dropped bytes    :                     0                     0 bps
     Low                 :                     0                     0 bps
     Medium-low          :                     0                     0 bps
     Medium-high         :                     0                     0 bps
     High                :                     0                     0 bps
Queue: 6, Forwarding classes: af3x
  Queued:
    Packets              :                     0                     0 pps
    Bytes                :                     0                     0 bps
  Transmitted:
    Packets              :                     0                      0 pps
    Bytes                :                     0                     0 bps
    Tail-dropped packets :                     0                     0 pps
    RED-dropped packets  :                     0                     0 pps
     Low                 :                     0                     0 pps
     Medium-low          :                     0                     0 pps
     Medium-high         :                     0                     0 pps
     High                :                     0                     0 pps
    RED-dropped bytes    :                     0                     0 bps
     Low                 :                     0                     0 bps
     Medium-low          :                     0                     0 bps
     Medium-high         :                     0                     0 bps
     High                :                     0                     0 bps
Queue: 7, Forwarding classes: null
  Queued:
    Packets              :                     0                     0 pps
    Bytes                :                     0                     0 bps
  Transmitted:
    Packets              :                     0                     0 pps
    Bytes                :                     0                     0 bps
    Tail-dropped packets :                     0                     0 pps
    RED-dropped packets  :                     0                     0 pps
```

```
     Low                 :                     0                     0 pps
     Medium-low          :                     0                     0 pps
     Medium-high         :                     0                     0 pps
     High                :                     0                     0 pps
    RED-dropped bytes    :                     0                     0 bps
     Low                 :                     0                     0 bps
     Medium-low          :                     0                     0 bps
     Medium-high         :                     0                     0 bps
     High                :                     0                     0 bps
```

Egress queue statistics are shown for all eight forwarding classes. Of significance here is that only one set of queues is displayed, despite there being two IFLs on the interface. Attempts to view per IFL stats display only local traffic:

```
{master}[edit]
jnpr@R1-RE0# run show interfaces queue xe-2/0/0.1
  Logical interface xe-2/0/0.1 (Index 333) (SNMP ifIndex 2978)
    Flags: SNMP-Traps 0x4000 VLAN-Tag [ 0x8100.1000 ]  Encapsulation: ENET2
    Input packets : 10
    Output packets: 13

{master}[edit]
jnpr@R1-RE0#
```

To view the port mode scheduling hierarchy, we quickly issue a VTY command on the MPC that houses the port mode CoS interface, which here is FPC 2:

```
NPC2(R1-RE0 vty)# sho cos scheduler-hierarchy

class-of-service EGRESS scheduler hierarchy - rates in kbps

---------------------------------------------------------------------------------
                                    shaping guarntd delaybf  excess
interface name               index    rate    rate    rate    rate      other
---------------------------- -----  ------- ------- ------- ------- -------------
xe-2/0/0                       148     5000       0       0       0
  q 0 - pri 0/1              60689     3000    1000       0      0%
  q 1 - pri 0/1              60689        0    1000       0      5%
  q 2 - pri 0/1              60689        0    1000       0     10%
  q 3 - pri 3/0              60689        0     500     10%      0%
  q 4 - pri 0/2              60689        0    1000       0     40%
  q 5 - pri 4/0              60689        0    2000   25000      0% exact
  q 6 - pri 0/1              60689        0    1000       0     30%
  q 7 - pri 2/5              60689        0       0       0      0%
. . .
```

The display confirms eight queues attached to the interface. No IFL-level queuing or information is shown. A later section details what all the scheduler queue-level settings mean; what is important now is that the two logical units on xe-2/2/0, unit 0 for bridges Layer 2 and unit 1 for routed L3, both share the same set of egress queues. There is no way to provide different levels of service for one IFL versus the other, or more importantly perhaps, to isolate one from the other should excess traffic levels appear, say as a result of a loop in the L2 network.

But all this soon changes as we segue into per unit scheduling. The output also confirms the IFD level shaping rate is in effect.

Per Unit Scheduler

Figure 5-21 illustrates the per unit scheduler mode of operation.

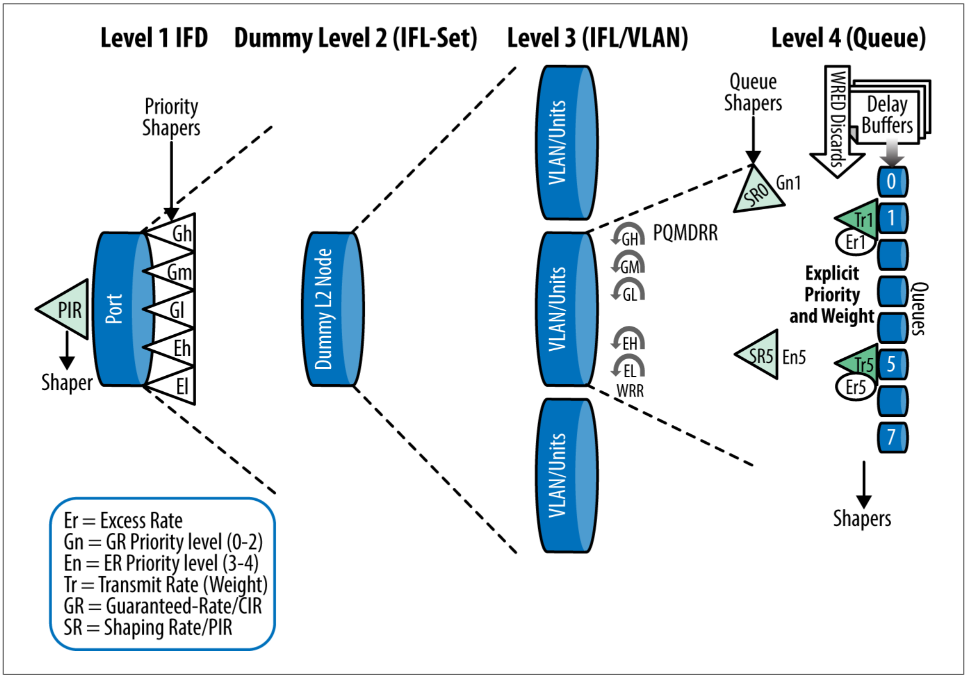


Figure 5-21. Trio Per-Unit Mode Scheduling: Queues for Each IFL.

In per unit scheduling, the system creates a level 3 hierarchy that supports a set of queues per IFL, or C-VLAN, rather than one set per port, as in the previous example. The added granularity lets you provide per IFL CoS, where some IFLs are shaped differently and perhaps even have altered scheduling behavior as it relates to priority, transmit, and delay buffer sizes. You evoke this scheduling mode by adding the `per-unit-scheduler` statement under an IFD that's housed in a queuing MPC.

> Only queuing MPCs support per unit scheduling in the current Trio hardware.

The configuration at R1 is modified to illustrate these concepts. In this example, the previous set of schedulers are adjusted to use the more common transmit rate as a percentage approach, rather than specifying an absolute bandwidth rate, as this allows for flexible scheduler usage over interfaces with wide ranging speeds or shaping rates. The change in scheduler rate from an absolute value to a percentage has no bearing on port level versus per unit or hierarchical CoS. The change could easily have been introduced in the previous port-level CoS example:

```
{master}[edit]
jnpr@R1-RE0# show class-of-service schedulers
sched_af4x_40 {
    transmit-rate percent 10;
    excess-rate percent 40;
    excess-priority high;
}
sched_af3x_30 {
    transmit-rate percent 10;
    excess-rate percent 30;
    excess-priority low;
}
sched_af2x_10 {
    transmit-rate percent 10;
    excess-rate percent 10;
    excess-priority low;
}
sched_af1x_5 {
    transmit-rate percent 10;
    excess-rate percent 5;
    excess-priority low;
}
sched_be_5 {
    transmit-rate percent 10;
    shaping-rate 3m;
    excess-priority low;
}
sched_nc {
    transmit-rate percent 10;
    buffer-size percent 10;
    priority high;
}
sched_ef_50 {
    transmit-rate {
        percent 40;
        rate-limit;
    }
    buffer-size temporal 25k;
    priority strict-high;
}
```

A queue's transmit rate percentage is based on the related IFL's shaping rate, when one is configured, or the IFD level shaping rate or port speed is used.

The changes that convert R1 from port level to per unit scheduling mode are pretty minor. In the class of service hierarchy, the scheduler map statement is removed from the applied to the IFD (note that the IFD-level TCP is left in place to shape the IFD), and a new TCP is defined for application to *both* of the IFLs on the xe-2/0/0 interface at the [edit class-of-service interface] hierarchy:

```
{master}[edit]
jnpr@R1-RE0# show class-of-service interfaces xe-2/0/0
output-traffic-control-profile TCP_PE-P_5;
unit 0 {
    output-traffic-control-profile tc-ifl;
    classifiers {
        ieee-802.1 ieee_classify;
    }
    rewrite-rules {
        ieee-802.1 ieee_rewrite;
    }
}
unit 1 {
    output-traffic-control-profile tc-ifl;
    classifiers {
        dscp dscp_diffserv;
    }
    rewrite-rules {
        dscp dscp_diffserv;
    }
}
```

Again, note that the scheduler map, which binds queues to schedulers, is now applied at the IFL level. This permits different scheduler maps and shaping rates on a per IFL basis; this example uses the same values for both units:

```
{master}[edit]
jnpr@R1-RE0# show class-of-service traffic-control-profiles
TCP_PE-P_5 {
    shaping-rate 5m;
}
tc-ifl {
    scheduler-map sched_map_pe-p;
    shaping-rate 1m;
}
```

The final change in switching from port to per unit mode occurs at the interfaces level of the hierarchy, where the per-unit-scheduler statement is added to the interface:

```
{master}[edit]
jnpr@R1-RE0# show interfaces xe-2/0/0
per-unit-scheduler;
vlan-tagging;
unit 0 {
    family bridge {
        interface-mode trunk;
        vlan-id-list 1-999;
    }
}
```

```
unit 1 {
    vlan-id 1000;
    family inet {
        address 10.8.0.0/31;
    }
    family iso;
}
```

In this example, the same TCP is applied to both IFLs, thereby giving them the same shaping and scheduling behavior. It can be argued that similar effects can be achieved with port-level operation by providing a shaped rate of 2 Mbps at the IFD level, which in this case matches the combined per unit shaping rates. However, per unit is providing several advantages over port based CoS; namely, with per unit:

You can have independent IFD and IFL shaping rates.

You can use different IFL-level shapers and scheduler maps to effect different CoS treatment.

Even if the same IFL level TCP is used, as in this example, per unit scheduling helps provide CoS isolation between the units that share an IFL. With the configuration shown, each IFL is isolated from traffic loads on the other; recall that in port mode all IFLs shared the same set of queues, scheduler, and shaped rate. As such, it's possible for excess traffic on unit 0, perhaps resulting from a Layer 2 malfunction that produces a loop, to effect the throughput of the Layer 3 traffic on unit 1. With per unit, each IFL is isolated to its shaped rate when isolates other IFLs from excessive loads.

After committing the change, the effects are confirmed:

```
jnpr@R1-RE0# run show class-of-service interface xe-2/0/0 detail
Physical interface: xe-2/0/0, Enabled, Physical link is Up
  Link-level type: Ethernet, MTU: 1518, LAN-PHY mode, Speed: 10Gbps, Loopback: None,
                    Source filtering: Disabled, Flow control: Enabled
  Device flags   : Present Running
  Interface flags: SNMP-Traps Internal: 0x4000
  Link flags     : Scheduler

Physical interface: xe-2/0/0, Index: 148
Queues supported: 8, Queues in use: 7
  Output traffic control profile: TCP_PE-P_5, Index: 28175
  Congestion-notification: Disabled

  Logical interface xe-2/0/0.0
    Flags: SNMP-Traps 0x24024000 Encapsulation: Ethernet-Bridge
    bridge
Interface       Admin Link Proto Input Filter         Output Filter
xe-2/0/0.0      up    up   bridge
Interface       Admin Link Proto Input Policer         Output Policer
xe-2/0/0.0      up    up
                           bridge

  Logical interface: xe-2/0/0.0, Index: 332
    Object                  Name                   Type                    Index
    Traffic-control-profile tc-ifl                 Output                  50827
    Rewrite                 ieee_rewrite            ieee8021p (outer)       16962
    Classifier              ieee_classify          ieee8021p               22868

  Logical interface xe-2/0/0.1
    Flags: SNMP-Traps 0x4000 VLAN-Tag [ 0x8100.1000 ]  Encapsulation: ENET2
    inet  10.8.0.0/31
    iso
    multiservice
Interface       Admin Link Proto Input Filter         Output Filter
xe-2/0/0.1      up    up   inet
                           iso
                           multiservice
Interface       Admin Link Proto Input Policer         Output Policer
xe-2/0/0.1      up    up
                           inet
                           iso
                           multiservice __default_arp_policer__

  Logical interface: xe-2/0/0.1, Index: 333
    Object                  Name                   Type                    Index
    Traffic-control-profile tc-ifl                 Output                  50827
    Rewrite                 dscp_diffserv          dscp                    23080
    Classifier              dscp_diffserv          dscp                    23080
```

In contrast to the per port mode, now each unit is listed with its own output TCP, which in this case contains the scheduler map that links the IFL to its own set of queues. This is further confirmed with the output of a `show interfaces queue` command, which now shows IFD-level aggregates as well as per IFL-level queuing statistics.

First the combined IFD level is verified. Only the EF class is shown to save space:

```
{master}[edit]
jnpr@R1-RE0# run show interfaces queue xe-2/0/0 forwarding-class ef
Physical interface: xe-2/0/0, Enabled, Physical link is Up
  Interface index: 148, SNMP ifIndex: 4373
Forwarding classes: 16 supported, 7 in use
Egress queues: 8 supported, 7 in use
Queue: 5, Forwarding classes: ef
  Queued:
    Packets              :               1493319                   445 pps
    Bytes                :             334496610                799240 bps
  Transmitted:
    Packets              :               1493319                    445 pps
    Bytes                :             334496610                799240 bps
    Tail-dropped packets :                     0                     0 pps
    RL-dropped packets   :                596051                   167 pps
    RL-dropped bytes     :             121594294                272760 bps
    RED-dropped packets  :                     0                     0 pps
     Low                 :                     0                     0 pps
     Medium-low          :                     0                     0 pps
     Medium-high         :                     0                     0 pps
     High                :                     0                     0 pps
    RED-dropped bytes    :                     0                     0 bps
     Low                 :                     0                     0 bps
     Medium-low          :                     0                     0 bps
     Medium-high         :                     0                     0 bps
     High                :                     0                     0 bps
```

And now, stats from each set of IFL queues, starting with the bridged Layer 2 IFL unit 0:

```
{master}[edit]
jnpr@R1-RE0# run show interfaces queue xe-2/0/0.0 forwarding-class ef
  Logical interface xe-2/0/0.0 (Index 332) (SNMP ifIndex 5464)
Forwarding classes: 16 supported, 7 in use
Egress queues: 8 supported, 7 in use
Burst size: 0
Queue: 5, Forwarding classes: ef
  Queued:
    Packets              :               1490691                   222 pps
    Bytes                :             333907938                399160 bps
  Transmitted:
    Packets              :               1490691                    222 pps
    Bytes                :             333907938                399160 bps
    Tail-dropped packets :                     0                     0 pps
    RL-dropped packets   :                586459                     0 pps
    RL-dropped bytes     :             119637526                     0 bps
    RED-dropped packets  :                     0                     0 pps
     Low                 :                     0                     0 pps
     Medium-low          :                     0                     0 pps
     Medium-high         :                     0                     0 pps
     High                :                     0                     0 pps
    RED-dropped bytes    :                     0                     0 bps
     Low                 :                     0                     0 bps
     Medium-low          :                     0                     0 bps
     Medium-high         :                     0                     0 bps
     High                :                     0                     0 bps
```

And now unit number 1, used for Layer 3 routing:

```
{master}[edit]
jnpr@R1-RE0# run show interfaces queue xe-2/0/0.1 forwarding-class ef
  Logical interface xe-2/0/0.1 (Index 333) (SNMP ifIndex 2978)
Forwarding classes: 16 supported, 7 in use
Egress queues: 8 supported, 7 in use
Burst size: 0
Queue: 5, Forwarding classes: ef
  Queued:
    Packets              :                  5679                   222 pps
    Bytes                :               1272096                398408 bps
  Transmitted:
    Packets              :                  5679                    222 pps
    Bytes                :               1272096                398408 bps
    Tail-dropped packets :                     0                     0 pps
    RL-dropped packets   :                 16738                  1319 pps
    RL-dropped bytes     :               3414552               2153288 bps
    RED-dropped packets  :                     0                     0 pps
     Low                 :                     0                     0 pps
     Medium-low          :                     0                     0 pps
     Medium-high         :                     0                     0 pps
     High                 :                     0                     0 pps
    RED-dropped bytes     :                     0                     0 bps
     Low                  :                     0                     0 bps
     Medium-low           :                     0                     0 bps
     Medium-high          :                     0                     0 bps
     High                 :                     0                     0 bps
```

The per unit scheduler hierarchy is displayed in the MPC:

```
NPC2(R1-RE0 vty)# show cos scheduler-hierarchy

class-of-service EGRESS scheduler hierarchy - rates in kbps
---------------------------------------------------------------------------------
                                    shaping guarntd delaybf  excess
interface name               index    rate    rate    rate    rate      other
---------------------------- -----  ------- ------- ------- ------- -------------
xe-2/0/0                       148     5000       0       0       0
  xe-2/0/0.0                   332     1000       0       0       0
    q 0 - pri 0/1            60689     3000     10%       0      0%
    q 1 - pri 0/1            60689        0     10%       0      5%
    q 2 - pri 0/1            60689        0     10%       0     10%
    q 3 - pri 3/0            60689        0     10%     10%      0%
    q 4 - pri 0/2            60689        0     10%       0     40%
    q 5 - pri 4/0            60689        0     40%   25000      0% exact
    q 6 - pri 0/1            60689        0     10%       0     30%
  xe-2/0/0.1                   333     1000       0       0       0
    q 0 - pri 0/1            60689     3000     10%       0      0%
    q 1 - pri 0/1            60689        0     10%       0      5%
    q 2 - pri 0/1            60689        0     10%       0     10%
    q 3 - pri 3/0            60689        0     10%     10%      0%
    q 4 - pri 0/2            60689        0     10%       0     40%
    q 5 - pri 4/0            60689        0     40%   25000      0% exact
    q 6 - pri 0/1            60689        0     10%       0     30%
  xe-2/0/0.32767               334        0    2000    2000       0
    q 0 - pri 0/1                2        0     95%     95%      0%
    q 3 - pri 0/1                2        0      5%      5%      0%
```

Note that now three sets of queues appear, one for each of the interface IFLs. The Layer 2 and Layer 3 IFLs, 0 and 1 respectively, show the same set of scheduling parameters, including their 1 Mbps shaping rates that stem from the common TCP applied to both. When desired, you can shape and schedule each IFL independently in per unit or H-CoS scheduling modes.

You may be wondering about the third set of queues. They are there to support the sending of LACP control traffic (as used to support AE link bonding) to the remote end of the link even when egress queues are congested. The `32767` unit is created automatically when VLAN tagging is in effect and a default scheduler map is automatically applied which supports 95%/5% BE and NC.

Per unit scheduling mode, which is supported only on queuing MPCs in the v11.4 release, is a logical step between the extremely coarse port-level mode and the upcoming hierarchical mode, which goes to the other extreme, with fine-grained queuing control at multiple levels of scheduling hierarchy.

Hierarchical Scheduler. Specifying the hierarchical-scheduler statement under a supported IFD provides full hierarchical CoS capabilities. Most of this chapter is devoted to H-CoS operation, so here it's sufficient to say that H-CoS add a new interface-set level construct that allows scheduling and shaping among a set of IFLs, each with their own set of queues that can be individually shaped. Refer to Figure 5-7 for an overview of hierarchical scheduling. Refer to Table 5-3 for details on current H-CoS scaling capabilities.

H-CoS and Aggregated Ethernet Interfaces

Aggregated Ethernet (AE) interfaces are quite common. Junos and Trio provide powerful CoS support over AE interfaces, including H-CoS scheduling mode in nonlink protection scenarios, albeit with the restrictions listed:

IFL-Sets are not supported.

Input CoS (input scheduler-map, input traffic-control-profile, input-shaping-rate) is not supported.

Dynamic interfaces are not supported as part of AE bundle; Demux interfaces are supported.

Aggregated Ethernet H-CoS Modes

An AE interface can operate in one of two modes when configured for H-CoS. These are referred to as the *scale* and the *replication* modes. The scale mode is also referred to as an equal division mode. The operating mode is determined by the setting of the member-link-scheduler parameter at the [edit class-of-service interface <ae-interface-name>] hierarchy.

```
{master}[edit]
jnpr@R1-RE0# set class-of-service interfaces ae0 member-link-scheduler ?
Possible completions:
  replicate            Copy scheduler parameters from aggregate interface
  scale                Scale scheduler parameters on aggregate interface
{master}[edit]
jnpr@R1-RE0# set class-of-service interfaces ae0 member-link-scheduler
```

By default, scheduler parameters are scaled using the equal division mode among aggregated interface member links. Figure 5-22 illustrates the key concepts of the equal division model.

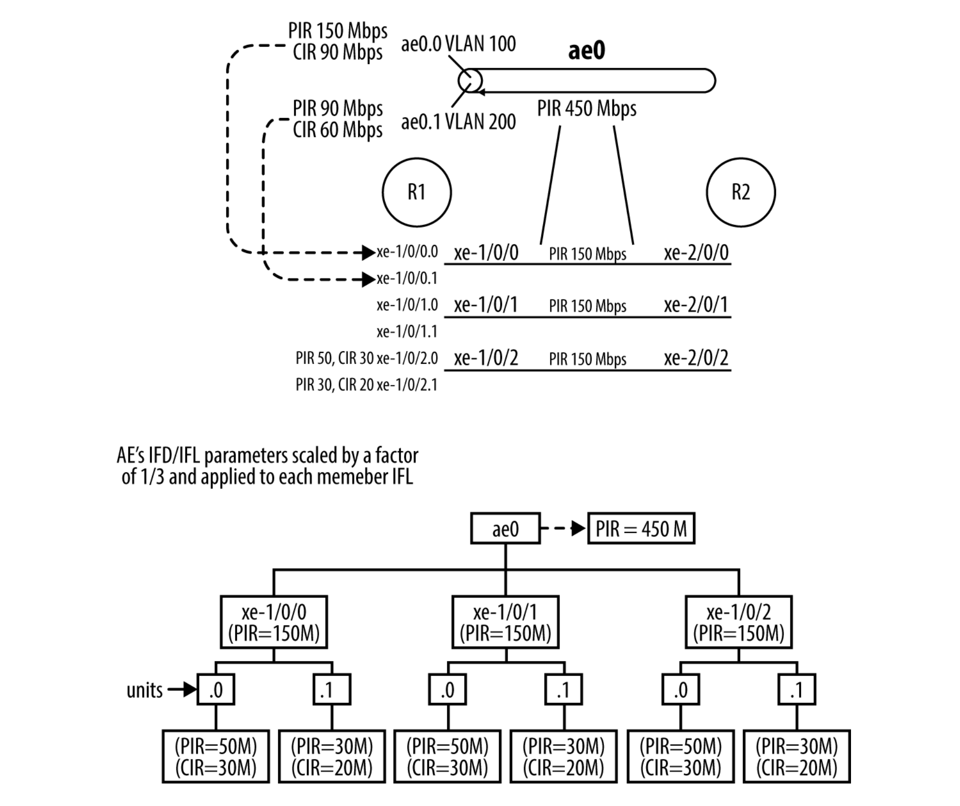


Figure 5-22. The Equal Share Mode of AE H-CoS.

In Figure 5-22, the two routers are connected by three links, which are bundled into an AE bundle called ae0 at R1. Two IFLs have been provisioned on the AE bundle: IFL 0 with VLAN 100 and IFL 1 with VLAN 200. A TCP with a shaping rate of 450 Mbps is applied to the AE0 IFD, while the PIR and CIR parameters shown are applied to the two IFLs, again through a TCP. Note that IFL 0 gets the higher CIR and PIR rates, being assigned 150 Mbps and 90 Mbps versus IFL 1's 90 and 60 Mbps values.

In the equal division mode, the IFD's shaping bandwidth is divided equally among all three member IFDs, netting each a derived PIR of 150 Mbps. Because traffic from the two VLAN can be sent over any of the three constituent links, each member link is given two logical units/IFLs, as shown, and then the AE bundle level's IFL CoS parameters are also divided by the number of member links and applied to the respective IFL created on those member links; thus the parameters from ae0.0 get scaled and applied to unit 0 of the member links while ae0.1 is scaled and applied to the unit 1 members. The resulting CoS hierarchy for the AE bundle in equal share mode is shown in ASCII art format to help illustrate the alternating nature of the way the CoS parameters are applied on a per unit basis.

```
                             ae0 --> PIR = 450 M
                           /  |  \
                          /   |   \
                  _______/    |    \_________
                  |           |             |
               xe-1/0/0       xe-1/0/1       xe-1/0/2
              (PIR=150M)      (PIR=150M)     (PIR=150M)
                 / \             / \              / \
                /   \           /   \            /   \
               /    |           |    \           |    ----+
              /     |           |     \          |        |
Subunits---> .0    .1           .0    .1        .0       .1
             /      |           |       \         \        \
       (PIR=50M)  (PIR=30M)  (PIR=50M) (PIR=30M) (PIR=50M) (PIR=30M)
       (CIR=30M)  (CIR=20M)  (CIR=30M) (CIR=20M) (CIR=30M) (CIR=20M)
```

In contrast, in the replication model, all the traffic control profiles and scheduler-related parameters are simply replicated from the AE bundle to the member links. There is no division of shaping rate, transmit rate, or delay buffer rate as is performed in the equal division model. ASCII art is, perhaps again, the best way to illustrate the mode differences.

```
                             ae0 --> PIR = 450 M
                           /  |  \
                          /   |   \
                  _______/    |    \_________
                  |           |             |
               xe-1/0/0       xe-1/0/0       xe-1/0/2
              (PIR=450M)      (PIR=450M)     (PIR=450M)
                 / \             / \              / \
                /   \           /   \            /   \
               /    |           |    \           |    ----+
              /     |           |     \          |        |
Subunits---> .0    .1           .0    .1        .0       .1
             /      |           |       \         \        \
       (PIR=150M) (PIR=90M)  (PIR=150M)(PIR=90M) (PIR=150M) (PIR=90M)
       (CIR=90M)  (CIR=60M)  (CIR=90M) (CIR=60M) (CIR=90M) (CIR=60M)
```

Note that now each link member is provided with the AE bundles IFD-level shaping rate, and the AE bundles IFL traffic parameters are applied to each member without any scaling.

Schedulers, Scheduler Maps, and TCPs

The queue level of the H-CoS hierarchy is configured by defining schedulers that are then linked into a scheduler map, which applies the set of schedulers to a given IFL, thus granting it queues. In contrast, a Traffic Control Profile (TCP) is a generic CoS container that can be applied at all points of the H-CoS hierarchy to affect CIR, PIR, and excess bandwidth handling. The TCP that is applied closest to the queue is special in that it also references a scheduler map. As shown previously for port mode CoS, the TCP with the scheduler map is applied at the IFD level. For per unit scheduling, the

TCP with a scheduler map is applied under each IFL. The primary change with H-CoS is the ability to also apply such a TCP to IFL-Sets at level 2.

As part of your basic CoS infrastructure, you will need to define at least one scheduler per forwarding class. Once the core is up and running, you can extend this to a multiservice edge by defining multiple schedulers with various queue handling characteristics. For example, to support various types of real-time media ranging from low-speed voice to HD video, you may want several EF schedulers to support rates, say 64 kbps, 500 kbps, 1 Mbps, 2 Mbps, 10 Mbps, etc. Then, based on the service that is being provisioned, a specific form of EF scheduler is referenced in the scheduler map. In effect, you build a stable of schedulers and then mix and match the set that is need based on a particular service definition.

Schedulers are defined at the `[editclass-of-serviceschedulers]` hierarchy and indicate a forwarding class's priority, transmit weight, and buffer size, as well as various shaping and rate control mechanisms.

```
{master}[edit class-of-service]
jnpr@R1-RE0# show schedulers
be_sched {
    transmit-rate percent 30;
    priority low;
    drop-profile-map loss-priority high protocol any drop-profile be_high_drop;
    drop-profile-map loss-priority low protocol any drop-profile be_low_drop;
}
ef_sched {
    buffer-size temporal 50k;
    transmit-rate percent 60 exact;
    priority high;
    drop-profile-map loss-priority high protocol any drop-profile ef_high_drop;
    drop-profile-map loss-priority low protocol any drop-profile ef_low_drop;
}
nc_sched {
    transmit-rate percent 10;
    priority low;
    drop-profile-map loss-priority high protocol any drop-profile nc_high_drop;
    drop-profile-map loss-priority low protocol any drop-profile nc_low_drop;
}
```

This example supports three forwarding classes—BE, EF, and NC—and each forwarding class's scheduler block is associated with a priority and a transmit rate. The transmit rate can be entered as a percentage of interface bandwidth or as an absolute value. You can rate limit (sometimes called *shape*) a queue with the `exact` keyword, which prevents a queue from getting any unused bandwidth, effectively capping the queue at its configured rate. Trio also supports hard policing rather than buffering using the `rate-limit` keyword. If there are not enough choices for you yet, alternatively, you can block a queue from using any excess bandwidth with excess priority none, with results similar to using rate limit.

In this example, the EF scheduler is set to high priority and is shaped to 60% of the interface speed, even when all other schedulers are idle, through the addition of the exact keyword. Using exact is a common method of providing the necessary forwarding class isolation when a high-priority queue is defined because it caps the total amount of EF that can leave each interface to which the scheduler is applied. As a shaper, this exact option does increase delays, however. When a low latency queue (LLQ) is desired, use the rate-limt option in conjunction with a temporally sized buffer.

With the configuration shown, each of the three forwarding classes are guaranteed to get at least their configured transmit percentages. The EF class is limited to no more than 60%, while during idle periods both the BE and NC classes can use 100% of egress bandwidth. When it has traffic pending, the high-priority EF queue is serviced as soon as possible—that is, as soon as the BE or NC packet currently being serviced has been completely dequeued.

Assuming a somewhat worst-case T1 link speed (1.544 Mbps), and a default MTU of 1,504 bytes, the longest time the EF queue should have to wait to be serviced is only about 7.8 milliseconds ($1/1.544^6$ * [1504 * 8]). With higher speeds (or smaller packets), the servicing delay becomes increasingly smaller. Given that the typical rule of thumb for the one-way delay budget of a Voice over IP application is 150 milliseconds, as defined in ITU's G.114 recommendation, this PHB can accommodate numerous hops before voice quality begins to suffer.

Scheduler Maps

Once you have defined your schedulers, you must link them a set of queues on an IFL using a scheduler-map. Scheduler maps are defined at the [editclass-of-servicescheduler-maps] hierarchy.

```
{master}[edit class-of-service]
jnpr@R1-RE0# show scheduler-maps
three_FC_sched {
    forwarding-class best-effort scheduler be_sched;
    forwarding-class expedited-forwarding scheduler ef_sched;
    forwarding-class network-control scheduler nc_sched;
}
```

Applying a scheduler-map to an interface places the related set of schedulers and drop profiles into effect. The older form of configuration places the scheduler map directly on the IFD or IFL; the former is shown here:

```
[edit class-of-service]
lab@Bock# show interfaces
fe-0/0/0 {
    scheduler-map three_FC_sched;
}
```

The newer and preferred approach is to reference the map within a TCP:

```
{master}[edit class-of-service]
jnpr@R1-RE0# show traffic-control-profiles
TCP_PE-P_5 {
    scheduler-map sched_map_pe-p;
    shaping-rate 5m;
}

{master}[edit class-of-service]
jnpr@R1-RE0# show interfaces xe-2/0/0
output-traffic-control-profile TCP_PE-P_5;
unit 0 {
    classifiers {
        ieee-802.1 ieee_classify;
. . .
```

Defining scheduler blocks that are based on a transmit percentage rather than an absolute value, such as in this example, makes it possible to apply the same `scheduler-map` to all interfaces without worrying whether the sum of the transmit rates exceeds interface capacity.

Why No Chassis Scheduler in Trio?

Because it is simply not needed, that's why. The `chassis-schedule-map` statement is designed for use on systems that were designed around four queues, when using IQ/IQ2EPICs, which offered support for eight queues. In these cases, the default behavior of the chassis scheduler when sending into the switch fabric was to divide the bandwidth into quarters, and sent one-quarter of the traffic over each of the four chassis queues (all at low priority). Because MX routers support eight queues, the default chassis scheduler does not need to be overridden, making this option irrelevant for MX platforms.

Configure WRED Drop Profiles. You configure a WRED drop profile at the `[editclass-of-servicedrop-profiles]` hierarchy. WRED drop profiles are placed into effect on an egress interface via application of a `scheduler-map`. Recall that, as shown previously, the `scheduler-map` references a set of schedulers, and each scheduler definition links to one or more drop profiles. It is an indirect process, to be sure, but it quickly begins to make sense once you have seen it in action.

Here are some examples of drop profiles, as referenced in the preceding `scheduler-map` example:

```
{master}[edit class-of-service]
jnpr@R1-RE0# showdrop-profiles
be_high_drop {
    fill-level 40 drop-probability 0;
    fill-level 50 drop-probability 10;
    fill-level 70 drop-probability 20;
}
be_low_drop {
    fill-level 70 drop-probability 0;
    fill-level 80 drop-probability 10;
}
ef_high_drop {
    fill-level 80 drop-probability 0;
```

```
        fill-level 85 drop-probability 10;
    }
    ef_low_drop {
        fill-level 90 drop-probability 0;
        fill-level 95 drop-probability 30;
    }
    nc_high_drop {
        fill-level 40 drop-probability 0;
        fill-level 50 drop-probability 10;
        fill-level 70 drop-probability 20;
    }
    nc_low_drop {
        fill-level 70 drop-probability 0;
        fill-level 80 drop-probability 10;
    }
```

In this example, the drop profiles for the BE and NC classes are configured the same, so technically a single-drop profile could be shared between these two classes. It's best practice to have per class profiles because ongoing CoS tuning may determine that a particular class will perform better with a slightly tweaked WRED threshold setting.

Both the BE and NC queues begin to drop 10% of high-loss priority packets once the respective queues average a 50% fill level. You can specify as many as 100 discrete points between the 0% and 100% loss points, or use the `interpolate` option to have all the points automatically calculated around any user-supplied thresholds. A similar approach is taken for the EF class, except it uses a less aggressive profile for both loss priorities, with discards starting at 80% and 90% fill for high and low loss priorities, respectively. Some CoS deployments disable RED (assign a 100/100 profile) for real-time classes such as EF, because these sources are normally UDP-based and do not react to loss in the same way that TCP-based applications do.

The `be_high` drop profile is displayed:

```
{master}[edit]
jnpr@R1-RE0# run show class-of-service drop-profile be_high_drop
Drop profile: be_high_drop, Type: discrete, Index: 27549
  Fill level    Drop probability
          40                   0
          50                  10
          70                  20
```

To provide contrast, the `be_high` profile is altered to use `interpolate`, which fills in all 100 points between 0% and 100% loss, as constrained by any user-specified fill/drop probability points:

```
{master}[edit]
jnpr@R1-RE0# show class-of-service drop-profiles be_high_drop
interpolate {
    fill-level [ 40 50 70 ];
    drop-probability [ 0 10 20 ];
}

{master}[edit]
```

```
jnpr@R1-RE0# run show class-of-service drop-profile be_high_drop
Drop profile: be_high_drop, Type: interpolated, Index: 27549
  Fill level    Drop probability
           0                   0
           1                   0
           2                   0
. . .
          51                  10
          52                  11
          54                  12
          55                  12
          56                  13
          58                  14
          60                  15
          62                  16
          64                  17
          65                  17
          66                  18
          68                  19
          70                  20
          72                  25
. . .
          96                  89
          98                  94
          99                  97
         100                 100
```

Scheduler Feature Support

Schedulers are a critical component of the Junos CoS architecture. Capabilities vary by hardware type. Table 5-13 highlights key capabilities and differences between Trio hardware and the previous I-chip-based IQ2 interfaces. In this table, the OSE PICs refer to the 10-port 10-Gigabit OSE PICs (described in some guides as the 10-Gigabit Ethernet LAN/WAN PICs with SFP+).

Table 5-12. Comparing Scheduler Parameters by PIC/Platform.

Scheduler Parameter	M320/T-Series	Trio MPC	IQ PIC	IQ2 PIC	IQ2E PIC	OSE on T-Series	Enhanced IQ PIC
Exact	Y	Y	Y	-	-	Y	Y
Rate limit	-	Y	-	Y	Y	Y	Y
Traffic Shaping	-	Y	-	-	Y	Y	Y
More than one H Pri Queue	Y	Y	Y	-	Y	-	Y
Excess Priority Sharing	-	Y	-	-	-	-	Y
H-CoS	-	Y (Q/EQ MPC)	-	-	Y	-	-

Traffic Control Profiles

As mentioned, a TCP is a CoS container that provides a consistent and uniform way of applying CoS parameters to portion of the H-CoS hierarchy. As of v11.4, TCP containers support the following options:

```
{master}[edit]
jnpr@R1-RE0# set class-of-service traffic-control-profiles test ?
Possible completions:
> adjust-minimum       Minimum shaping-rate when adjusted
+ apply-groups         Groups from which to inherit configuration data
+ apply-groups-except  Don't inherit configuration data from these groups
> delay-buffer-rate    Delay buffer rate
> excess-rate          Excess bandwidth sharing proportion
> excess-rate-high     Excess bandwidth sharing for excess-high priority
> excess-rate-low      Excess bandwidth sharing for excess-low priority
> guaranteed-rate      Guaranteed rate
> overhead-accounting  Overhead accounting
  scheduler-map        Mapping of forwarding classes to packet schedulers
> shaping-rate         Shaping rate
> shaping-rate-excess-high  Shaping rate for excess high traffic
> shaping-rate-excess-low  Shaping rate for excess low traffic
> shaping-rate-priority-high  Shaping rate for high priority traffic
> shaping-rate-priority-low  Shaping overhead-accounting  rate for low priority traffic
> shaping-rate-priority-medium  Shaping rate for medium priority traffic
{master}[edit]
```

The `scheduler-map` statement in a TCP is only used for the TCP that is applied closest to the queues. For H-CoS and per unit scheduler modes, this is typically the IFL level. TCPs applied at the IFD and IFL-Set levels normally only contain shaped rates, guaranteed rates, and excess bandwidth sharing settings. The use of the `delay-buffer-rate` option to set a reference bandwidth for use by a child node or queue was discussed previously.

Overhead Accounting on Trio. The `overhead-accounting` option in a TCP is used to alter how much overhead is factored into shaping and G-Rates, and is designed to accommodate different encapsulations, such as frame to cell for ATM-based B-RAS/DSLAM applications, or to simply account for the presence of one versus two VLAN tags in a stacked/Q-in-Q environment. Cell and frame modes are supported, with the default being frame mode.

As an example, consider that a B-RAS aggregator may receive untagged frames from its subscribers while the MX, upstream, receives dual-tagged traffic resulting in eight extra bytes being added to the frame, at least from the viewpoint of the subscriber, who likely sent untagged traffic to begin with. In this case the overhead accounting function is used to remove eight bytes (–8) from the accounting math, such that the end user's realized bandwidth more closely matches the service definition and reflects the (untagged) traffic that user actually sends and receives, thereby avoiding any indirect penalty that stems from the network's need to impose VLAN tags on a per service and subscriber basis.

The available range is –120 through 124 bytes. The system rounds up the byte adjustment value to the nearest multiple of 4. For example, a value of 6 is rounded to 8, and a value of –10 is rounded to –8.

Trio differs from IQ2 interfaces in that by default it factors Ethernet Layer 1 overhead, including 20 bytes for the preamble and interframe gaps, as well as the Ethernet frame overhead of 18 bytes, as used for MAC addresses, type code, and FCS. Thus, the default shaping overhead for Trio is 38 bytes per frame. To remove the preamble and IPG from the calculator, subtract 20 bytes:

```
{master}[edit]
jnpr@R1-RE0# show class-of-service traffic-control-profiles tc-ifd
overhead-accounting frame-mode bytes -20;
```

And, to confirm the change TCP is displayed in operational CLI mode:

```
{master}[edit]
jnpr@R1-RE0# run show class-of-service traffic-control-profile tc-ifd
Traffic control profile: tc-ifd, Index: 50819
  Scheduler map: <default>
  Overhead accounting mode: Frame Mode
  Overhead bytes: -20
```

Trio Scheduling and Priority Summary

This section detailed Trio scheduling modes and behavior, including priority demotion at both queues and scheduler nodes, as well as the default behavior and configuration options for scheduler burst sizes and queue/scheduler node delay bandwidth buffers. Excess bandwidth sharing, and how this relates to an interfaces mode, as either PIR or PIR/CIR, was also dicussed.

MX Trio CoS Defaults

Junos software comes with a set of default CoS settings that are designed to ensure that both transit and control plane traffic is properly classified and forwarded. This means all Juniper boxes are IP CoS enabled, albeit at a low level of functionality, right out of the box, so to speak. You will want to modify these defaults to tailor behavior, gain support for additional forwarding classes, and to ensure consistent classification and header rewrite operations throughout your network. A summary of default CoS characteristics includes the following:

> Support for two forwarding classes (BE and NC) and implements for an IP precedence-style BA classifier that maps network control into queue 3 while all other traffic is placed into queue 0 as BE.

A scheduler is placed into effect on all interfaces that allocates 95% of the bandwidth to queue 0 and the remaining 5% to queue 3. Both of the queues are low-priority, which guarantees no starvation in any platform.

A default WRED profile with a single loss point is placed into effect. The 100% drop at 100% fill setting effectively disables WRED.

No IP packet rewrite is performed with a default CoS configuration. Packets are sent with the same markers as when they were received.

No MPLS EXP or IEEE802.1p rewrites. The former is a departure from CoS defaults on M/T series platforms, which have a default EXP rewrite rule in effect that sets EXP based on queue number (0 to 3) and PLP.

Per port scheduling is enabled, no shaping or guaranteed rates are in effect, and IFD speed is the factor for all bandwidth and delay buffer calculations.

Four Forwarding Classes, but Only Two Queues

The default CoS configuration defines four forwarding classes—BE, EF, AF, and NC—that are mapped to queues 0, 1, 2, and 3, respectively. However, as noted previously, there is no default classification that will result in any traffic being mapped to either the AF or the EF class. This is good, because as also noted previously, no scheduling resources are allocated to queue 1 or 2 in a default CoS configuration. Some very interesting and difficult-to-solve problems occur if you begin to classify AF or EF traffic without first defining and applying schedulers for those classes. Doing so typically results in intermittent communications (some small trickle credit is given to 0% queues to prevent total starvation, along the lines of two MTUs worth of bandwidth and buffer) for the AF/EF classes, and this intermittency is tied to the loading levels of the BE and NC queues given that when there is no BE or NC traffic, more AF/EF can be sent, despite the 0% default weighting.

```
{master}[edit]
jnpr@R1-RE0# show class-of-service

{master}[edit]
jnpr@R1-RE0#
```

With no CoS configuration present, the default FCs and schedulers are shown:

```
{master}[edit]
jnpr@R1-RE0# run show class-of-service forwarding-class
Forwarding class         ID   Queue  Restricted  Fabric     Policing   SPU
                                      queue       priority   priority   priority
  best-effort             0    0       0           low        normal     low
  expedited-forwarding    1    1       1           low        normal     low
  assured-forwarding      2    2       2           low        normal     low
  network-control         3    3       3           low        normal     low
{master}[edit]
jnpr@R1-RE0# run show class-of-service scheduler-map
Scheduler map: <default>, Index: 2

  Scheduler: <default-be>, Forwarding class: best-effort, Index: 21
```

```
      Transmit rate: 95 percent, Rate Limit: none, Buffer size: 95 percent,
        Buffer Limit: none, Priority: low
      Excess Priority: low
      Drop profiles:
        Loss priority   Protocol    Index    Name
        Low             any             1    <default-drop-profile>
        Medium low      any             1    <default-drop-profile>
        Medium high     any             1    <default-drop-profile>
        High            any             1    <default-drop-profile>

    Scheduler: <default-nc>, Forwarding class: network-control, Index: 23
      Transmit rate: 5 percent, Rate Limit: none, Buffer size: 5 percent, Buffer Limit:
  none, Priority: low
      Excess Priority: low
      Drop profiles:
        Loss priority   Protocol    Index    Name
        Low             any             1    <default-drop-profile>
        Medium low      any             1    <default-drop-profile>
        Medium high     any             1    <default-drop-profile>
        High            any             1    <default-drop-profile>
```

Recognizing CoS Defaults Helps Spot Errors and Mistakes

Anytime you see a scheduler with 95%/5% for queues 0 and 3, it's a really good indication you are dealing with a default CoS configuration. When this is unexpected, check to make sure the CoS stanza or the interface within the CoS stanza is not deactivated. If all else fails and the configuration is correct, check the `cosd` or messages log while committing the configuration (consider using commit full as well). In some cases, CoS problems, either relating to unsupported configurations or lack of requisite hardware, are not caught by the CLI, but are reported in the log. In some cases, an error results in the configured CoS parameter being ignored, in which case the interface gets the default CoS setting for some or all parameters.

Default BA and Rewrite Marker Templates

Junos creates a complete set of BA classifiers and rewrite marker tables for each supported protocol family and type, but most of these tables are not used in a default CoS configuration. For example, there is both a default IP precedence (two actually) and a default DSCP classifier and rewrite table. You can view default and custom tables with the `showclass-of-serviceclassifier` or `showclass-of-servicerewrite-rule` command.

The default values in the various BA classifier and rewrite tables are chosen to represent the most common/standardized usage. In many cases, you will be able to simply apply the default tables. Because you cannot alter the default tables, it is suggested that you always create custom tables, even if they end up containing the same values as the default table. This is not much work, given that you can copy the contents of the default tables into a custom table, and in the future you will be able to alter the customer tables as requirements change. For example, to apply the default EXP rewrite rules include the `rewrite-rules exp default` statement at the `[edit class-of-service interfaces interface-name unit logical-unit-number]` hierarchy level.

In a default configuration, input BA classification is performed by the `ipprec-compatibility` table and no IP rewrite is in effect, meaning the CoS marking of packets at egress match those at ingress.

```
{master}[edit]
jnpr@R1-RE0# run show class-of-service interface xe-2/0/0
Physical interface: xe-2/0/0, Index: 148
Queues supported: 8, Queues in use: 4
Total non-default queues created: 8
  Scheduler map: <default>, Index: 2
  Congestion-notification: Disabled

  Logical interface: xe-2/0/0.0, Index: 332

  Logical interface: xe-2/0/0.1, Index: 333
    Object                 Name                  Type           Index
    Classifier             ipprec-compatibility  ip                13

. . .
```

The output from this IP- and family bridge-enabled interface confirms use of the default scheduler map and the absence of any rewrite rules. Note the default IP precedence-based classifier is in effect on the Layer 3 unit; in contrast, the Layer 2 bridge unit has no default IEEE802.1p classifier.

ToS Bleaching?

ToS bleaching is a term used to describe the resetting, or normalization, of ToS markings received from interfaces. Generally, unless you are in some special situation where paranoia rules, the concept of an untrusted interface is limited to links that connect other networks/external users. Given that by default all Junos IP interfaces have an IP precedence classifier in effect, it is a good idea to at least prevent end users from sending traffic marked with CS6 or CS7 to prevent them from getting free bandwidth from the NC queue, in the best case, or at the worst, trying to get free bandwidth and in so doing congesting the NC queue, which can lead to control plane flaps and all the joy that brings.

You can use a MF classifier to reset ToS markings if you need to support more than one FC. For IFLs that are relegated to only one FC, the best practice is fixed classification. This statement forces all traffic received on xe-2/2/0.0 into the BE forwarding class; at egress ToS rewrite bleaches the DSCP field (which subsumes IP precedence) to the specified setting, which for BE is normally all 0s:

```
{master}[edit class-of-service]
jnpr@R1-RE0# set interfaces xe-2/2/0 unit 0 forwarding-class be
```

If your network uses MPLS, you should add an explicit MPLS rewrite rules to all core-facing interfaces to ensure predictable and consistent MPSL EXP rewrite, which is then used by downstream P-routers to correct classify incoming MPLS traffic. Failing to do this on MX Trio platforms can lead to unpredictable classification behavior as currently, the IP precedence bits are written into the EXP field when an EXP rewrite rules is not in place.

MX Trio CoS Defaults Summary

This section detailed the factor default behavior of Trio-based MX MPC when no explicit CoS configuration is in effect. As always, things can evolve so it's best to check the documentation for your release, as the defaults described here are based on v11.4R1.

Always make sure you back up any user-defined FC with a scheduler, and be sure to include that scheduler in the scheduler map applied to any interface that is expected to transport that FC. Placing traffic into an undefined/default FC results a pretty effective blackhole as only minimal resources are provided to such traffic.

Predicting Queue Throughput

It took some time to get here. CoS is a big subject, and the Trio capabilities are so broad in this regard that it can be overwhelming at first, even if you are already familiar with the general CoS processing and capabilities of Junos platforms. This section is designed to serve as a practical review of the key points and behaviors covered. The approach taken here is somewhat pop quiz-like to keep things fun.

If you find yourself surprised at some of the answers, read back over the last 100 pages or so. The truth is out there.

CoS is one of those tricky subjects; the kind where everything is working fine, and so, feeling bored, you decide to make a small tuning adjustment, for example explicitly assigning an excess rate to a queue that is equal to the one it has already been using through default inheritance. Yes, a very minor, very small, seemingly innocuous change. Yet boom! Suddenly all sorts of behaviors change and you are once again eternally thankful for the `rollback` feature of Junos. It's hard to test CoS in a live network for numerous reasons that are so obvious they need not be enumerated here. The takeaway is that you should test and model the behavior of any proposed CoS change in a lab setting to make sure you understand all behavior changes before *rolling* the proposed change into production. Small changes really can have big impacts, and these are often unanticipated, as the material in this section is intended to demonstrate.

Where to Start?

Before jumping into the details, here's a brief review of some important points to keep in mind:

- The IFL shaping rate limits queue throughput. The IFD shaping rate limits IFL throughput.
- In PIR mode, transmit rate is based on shaped speed. A queue with a 10% transmit rate on an IFL shaped to 10 Mbps can send up to 10% of shaped speed at normal priority (H, M, L) and is then entitled to another 10% of any remaining shaped bandwidth with default excess rate settings.
- A queue transmit rate is a guarantee, of sorts, assuming you design things correctly. But this is not the same as guaranteed rate/G-Rate. The latter is a function of scheduler nodes and use of `guaranteed rate` in a TCP. The former is a queue-level setting only. It's possible for a GL queue to be in excess, for example because it has exceeded a low transmit rate value, while the L3 scheduler node for that IFL has excess G-Rate capacity and therefore sends the traffic as guaranteed via the priority promotion process. The key here is that such a queue thinks it's in excess, and so a low excess rate setting, or blocking excess by setting excess none, can reduce its throughput, even though the attached scheduler has remaining G-Rate.
- By default, transmit and excess rates are equal. In CIR mode, the transmit rate is based first on CIR/guaranteed rate, and then when in excess on the PIR/shaped rate. On a 2 Mbps CIR/10 Mbps PIR IFL, a queue with 10% transmit rate gets 10% of 2 Mbps at its guaranteed priority level (H, M, or L) and is also able to get 10% (by default) of the remaining PIR (8 Mbps) at its excess level (EH or EL).
- In CIR mode, if you have a lot of PIR bandwidth, it tends to favor queues with high excess rates and priority. The opposite is also true, in that a configuration with a large CIR tends to favor queues with high transmit rates. Here, PIR is the difference between IFL-shaped speed and the configured guaranteed rate, and not the difference between the sum of queue transmit weights and the IFL shaping speed. An IFL shaped to 10 Mbps with a 5 Mbps CIR has 5 Mbps of G-Rate and 5 Mbps of PIR, regardless of how many queues you have or if the queue's percentages are underbooked (i.e., summing to less than 100%).
- By default, excess weighting is based on transmit rate. If you explicitly set an excess rate for one queue, you place the IFL into the excess rate mode and all queues not explicitly set with an excess rate gets a minimal weight of 1. If you set excess rate on one queue, it's a good idea to set it on all. Many users don't anticipate the effects of excess rate mode; you can always explicitly set minimum weights if that is what you want, and at least the configuration makes it clear what to expect.
- A queue's throughput is generally determined by its priority, transmit rate, excess priority, and excess rate, in that order, but this is not always so. A queue with a low transmit rate but with a very high excess rate can get more bandwidth then a higher priority queue that has a higher transmit rate if the PIR-to-CIR ratio is large.

For example, setting excess none prevents even a high-priority queue from getting any excess bandwidth, and if your CIR model has a lot of excess bandwidth, well, there you go.

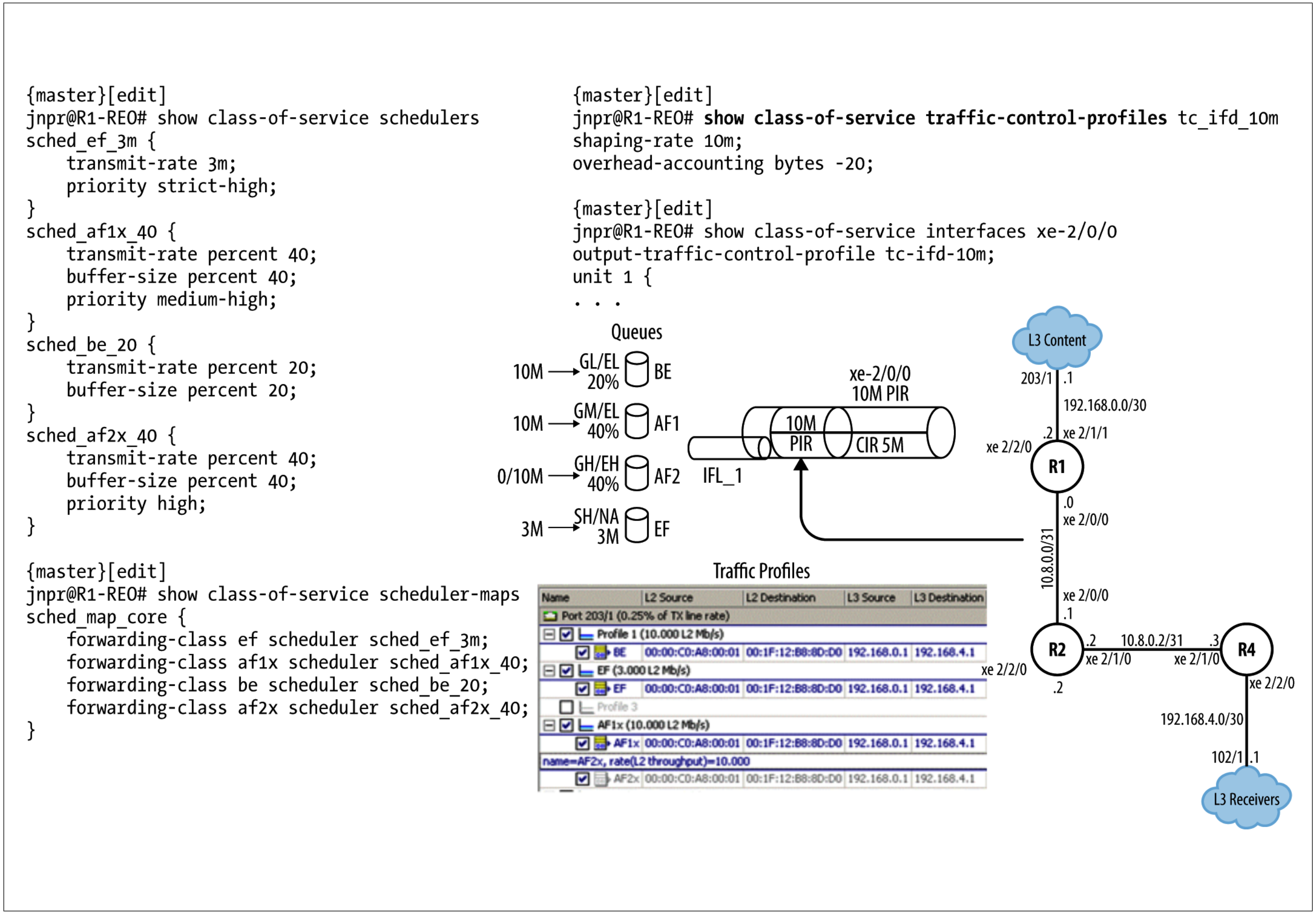
- A queue can demote GH, GM, or GL to its configured excess rate when it exceeds the configured transmit rate. By default, SH/H gets EH while all others get EL. SH is never demoted at the queue level when configured properly as rate limiting/shaping should be in effect.
- SH gets 100% of the IFL-shaped rate and cannot be demoted; the transmit rate for an SH queue is only relevant for shaping or rate limiting. You must have some form of rate control if you deploy SH. Adding a transmit rate as percentage or as absolute bandwidth to SH is primarily done for cosmetic reasons, unless you are using `rate-limit` or `exact` to limit/shape the queue, in which case the limits are based on the specified rate.
- Anytime you deploy SH, you are in effect overbooking the interface. Again, the assumption is that you will rate limit the EF queue, but still this can make the math, and results, less than intuitive. For example, once you subtract the SH traffic from the G-Rate, it's possible that a queue set to a GL priority with a 10% rate on a 10 Mbps CIR mode interface will enter the excess region long before it sends the 1 Mbps of GL traffic it was configured for. Stay tuned for details.
- In per unit mode, CIR cannot be overbooked. Watch for warnings in the logs or look to confirm that all guaranteed rates are in fact programmed in the PFE. If an interface cannot get its G-Rate, it gets a G-Rate of 0, and even if the underlying shaping rate is increased to accommodate its needs you may have to flap that interface before it will get the configured G-Rate.
- Be aware of the 2 Mbps G-Rate preassigned to the LACP control function on all VLAN-tagged interfaces. In H-CoS mode, the ability to overbook means you don't have to increase underlying shaping rates to accommodate this scheduler's 2 Mbps of G-Rate bandwidth. In per unit mode, failing to do so can lead to IFLs with a 0 G-Rate. Having no G-Rate is not so bad, as long as all the other IFLs on that IFD also lack one. However, if two IFLs are intended to contend in a predictable manner based on the ratio of their G-Rate, and one ends up with no G-Rate, then all bets are truly off.
- CoS schedulers are not permitted on IRB interfaces, but don't forget to put BA or MF classifiers there. Traffic going through an IRB is replicated, and any ingress classification is lost in this process. This is true for both directions, L2 to L3, and L3 back into L2.
- There is no command, CLI or shell, to tell if a queue is currently at normal versus excess priority. Such changes can occur almost instantaneously, so it's not clear that such a command is useful for real-time analysis, but the history of such a command could prove if, and how often, the queue enters excess. One of the best ways to prove or disprove excess usage is to set the queue to H/MH, and then set excess none (changing the priority can have side effects, but excess none is not

supported with GL, the default priority). If the queue throughput drops off, it's a good indication it was using excess bandwidth.

- Because of priority promotion and demotion, a GL queue can be below its transmit rate and be in excess. In contrast, a GH/GM is never in excess until it has met transmit rate, as these priorities cannot be demoted based on G-Rate. In like fashion, a GL queue can be above its transmit and be promoted, such that it is in the guaranteed region. If the sum of GH + GM exceed the IFL's G-rate, GL is demoted and the excess needed to meet GH/GM transmit rate is taken from the PIR, which means less excess to share.
- Per priority shapers can demote any priority except GL. If a queue is set to 10 Mbps of GH and you have a 5 Mbps priority high shaper in effect, don't be surprised when the queue still gets 10 Mbps. It's just that 5 Mbps of that is now sent at EH due to demotion at the per priority shaper. A per priority shaper is a good way to limit the volume of a given priority, but not really effective at limiting queue throughput.
- In v11.4R1, a SH scheduler with a rate limit bases its queue's transmit rate against the PIR, even when a CIER is configured. Other priorities, or SH when you do not include `rate-limit` or `exact`, use the CIR to compute transmit speeds. Depending on the difference between CIR and shaped/PIR rate, this can have a big impact on EF bandwidth allocation. This behavior is only noted when transmit is a percentage. If you specify an absolute bandwidth, that is the value assigned.

Trio CoS Proof-of-Concept Test Lab

Figure 5-23 details a simplified lab topology that facilitates analysis of Trio CoS behavior.



```
{master}[edit]
jnpr@R1-RE0# show class-of-service schedulers
sched_ef_3m {
    transmit-rate 3m;
    priority strict-high;
}
sched_af1x_40 {
    transmit-rate percent 40;
    buffer-size percent 40;
    priority medium-high;
}
sched_be_20 {
    transmit-rate percent 20;
    buffer-size percent 20;
}
sched_af2x_40 {
    transmit-rate percent 40;
    buffer-size percent 40;
    priority high;
}
```

```
{master}[edit]
jnpr@R1-RE0# show class-of-service scheduler-maps
sched_map_core {
    forwarding-class ef scheduler sched_ef_3m;
    forwarding-class af1x scheduler sched_af1x_40;
    forwarding-class be scheduler sched_be_20;
    forwarding-class af2x scheduler sched_af2x_40;
}
```

```
{master}[edit]
jnpr@R1-RE0# show class-of-service traffic-control-profiles tc_ifd_10m
shaping-rate 10m;
overhead-accounting bytes -20;
```

```
{master}[edit]
jnpr@R1-RE0# show class-of-service interfaces xe-2/0/0
output-traffic-control-profile tc-ifd-10m;
unit 1 {
. . .
```

Name	L2 Source	L2 Destination	L3 Source	L3 Destination
Port 203/1 (0.25% of TX line rate)				
⊟ ☑ Profile 1 (10.000 L2 Mb/s)				
☑ BE	00:00:C0:A8:00:01	00:1F:12:B8:8D:D0	192.168.0.1	192.168.4.1
⊟ ☑ EF (3.000 L2 Mb/s)				
☑ EF	00:00:C0:A8:00:01	00:1F:12:B8:8D:D0	192.168.0.1	192.168.4.1
□ Profile 3				
⊟ ☑ AF1x (10.000 L2 Mb/s)				
☑ AF1x	00:00:C0:A8:00:01	00:1F:12:B8:8D:D0	192.168.0.1	192.168.4.1
name=AF2x, rate(L2 throughput)=10.000				
☑ AF2x	00:00:C0:A8:00:01	00:1F:12:B8:8D:D0	192.168.0.1	192.168.4.1

Figure 5-23. CoS Test Lab Topology.

In this example, a single IFL is defined that uses four queues (0, 1, 2, and 5). The IFD remains shaped at 10 Mbps throughout the experiment, and the ingress traffic loads are based on 300 byte Layer 2 frames at the rates shown. The total ingress load is either 23 Mbps or 33 Mbps, which given the 10 Mbps shaping at egress from the xe-2/0/0 interface at R1 is guaranteed to induce congestion. This section focuses on queuing and scheduling behavior; to the focus, only IP traffic is flowing. This traffic arrives at R1 for classification and then egresses toward R2 after being scheduled and shaped. It's this egress behavior at R1 that serves as the area of focus in this section.

There is no IGP running, which is good, as no NC queue is provisioned for this test; a static route at R1 for the 192.168.4.0/30 destination network ensures it knows what to do with the test traffic and that we are immune from control plane flap and the resulting test disruption that results.

The figure also shows a basic set of scheduler definitions, the scheduler map, as well as the TCP used to shape the interface.

The IFD-level TCP used for shaping is adjusted to subtract 20 bytes of overhead from its calculations. This eliminates the Layer 1 overhead and matches the router to the tester, as the latter generates traffic based on L2, not L1, rates.

Most of this figure remains in place as various scenarios are tested. As such, let's take a closer look at the schedulers that are configured. In the majority of cases, the AF2 queue consumes no bandwidth as it usually has no input load offered; in some cases, a 10 Mbps input rate may be started.

The EF queue is a special case given its use of a SH scheduling priority. This queue always gets served first and is only limited by the input traffic rate; recall that as SH, it has 100% transmit rate (the configured rate does not really matter to a SH queue, except for rate limiting/shaping) and it cannot be demoted at the queue level (it's limited), nor at scheduler nodes based on G-Rate given its H priority. The only thing that prevents this queue from starving all other traffic in this configuration is the input limit of 3 Mbps that is imposed by the traffic generator. *Without a rate limit or exact shaping, or an ingress policer to limit, this would be a very dangerous configuration to deploy, notwithstanding the lack of a dedicated NC queue to boot!*

Given AF2's usual lack of input, and the fixed behavior of the EF queue, it's clear that most of the analysis is directed to the interaction of the BE and AF1 queues as they jockey for their transmit rates and a share of any remaining bandwidth.

The sum of queue transmit rates equals 100% (not counting EF's 100%, of course), and all queues except EF are excess eligible. With no explicit excess rate configurations, the defaults are in place, making this a good place to begin the Trio CoS Proof of Concept (PoC) testing.

A Word on Ratios

Before moving into specifics, it's good to know ratios as they are used to proportion both guaranteed and excess bandwidth. Taking Q0 and Q1 as examples, they have a 20:40 ratio between their transmit rates. That simplifies to a 1:2 ratio. Speaking in terms of rounded numbers, the ratio for queue 0 is therefore 1:3 or 0.333, whereas the ratio for queue 1 is 2:3 or 0.666. If there was 1 Mbps to split between these queues, then Q0 gets 1 * 0.333, or 0.333 Mbps, while queue 1 gets 1 * 0.666, or 0.666 Mbps. Summing the two, you arrive at the total 1 Mbps that was available to be shared.

Example 1: PIR Mode

Pop Quiz 1:

How will excess bandwidth be shared?

What does a transmit rate of 20% mean in this context? Will it be 2 Mbps or 1 Mbps?

What throughput do you expect for the BE queue?

Details on the scheduler and its settings are displayed:

```
NPC2(R1-RE0 vty)# sho cos scheduler-hierarchy

class-of-service EGRESS scheduler hierarchy - rates in kbps
---------------------------------------------------------------------------------
                                    shaping guarntd delaybf  excess
interface name                index   rate    rate    rate    rate      other
---------------------------- -----  ------- ------- ------- ------- -------------
xe-2/0/0                       148    10000       0       0       0
  xe-2/0/0.1                   325    10000       0       0       0
    q 0 - pri 0/0            20205        0     20%     20%      0%
    q 1 - pri 2/0            20205        0     40%     40%      0%
    q 2 - pri 3/0            20205        0     40%     40%      0%
    q 5 - pri 4/0            20205        0    3000       0      0%
  xe-2/0/0.32767               326        0    2000    2000       0
    q 0 - pri 0/1                2        0     95%     95%      0%
    q 3 - pri 0/1                2        0      5%      5%      0%

NPC2(R1-RE0 vty)# sho cos halp ifl 325
IFL type: Basic

--------------------------------------------------------------------------------
IFL name: (xe-2/0/0.1, xe-2/0/0)   (Index 325, IFD Index 148)
    QX chip id: 0
    QX chip dummy L2 index: -1
    QX chip L3 index: 3
    QX chip base Q index: 24
Queue    State         Max       Guaranteed   Burst  Weight Priorities Drop-Rules
Index                  rate         rate      size            G    E   Wred  Tail
------ ----------- ----------- ------------ ------- ------ ---------- ----------
    24  Configured    10000000      2000000  131072    153   GL    EL     4     9
    25  Configured    10000000      4000000  131072    307   GM    EL     4   138
    26  Configured    10000000      4000000  131072    307   GH    EH     4   202
```

```
27  Configured    10000000          0  131072      1   GL    EL     0   255
28  Configured    10000000          0  131072      1   GL    EL     0   255
29  Configured    10000000   Disabled  131072    230   GH    EH     4   196
30  Configured    10000000          0  131072      1   GL    EL     0   255
31  Configured    10000000          0  131072      1   GL    EL     0   255
```

The output goes far in answering the questions. It's clear that transmit rates are based on IFD shaping rate (this is PIR, so there is no G-Rate), and that excess bandwidth is shared based on the queues transmit rate. Hence, BE at 20% gets one-half of the weighting of the AF classes, which are both at 40%. The scheduler settings confirm that the IFL does not have a guaranteed rate configured; G-Rates are not supported at the IFD level. This confirms a PIR mode interface example.

Pop Quiz 1 Answers:

How will excess bandwidth be shared?

Excess is shared based on priority and excess weight. Queue 1 has its default excess priority modified such that both queue 0 and 1 have the same excess priority, allowing them to share according to weight. Excess weight is also at the default, which means weighting ratio is based on queue transmit weight. At one point, only CIR mode interfaces could be configured with excess rate parameters, as the lack of G-Rate in PIR mode means all traffic is in excess of the 0 G-Rate used on PIR. In v11.4, you can configure excess rate for PIR mode; this example shows usage of the default parameter values.

What does a transmit rate of 20% mean in this context? Will it be 2 Mbps or 1 Mbps?

In PIR mode, the lesser of the IFD speed/shaping rate or IFL shaping rate is used to calculate transmit rate values as bandwidth. In this case, the calculation is based on the 10 Mbps IFD shaping rate; hence 20% yields 2 Mbps.

What throughput do you expect for the BE queue?

This is where the rubber meets the road, so to speak. The math:

10 Mbps PIR available

EF gets 3 M, 7 Mbps PIR remains

AF1 gets 4 Mbps of transmit based on MH priority/rate, 3 Mbps of PIR remains

BE gets 2 Mbps of transmit based on L priority weight, 1 Mbps of PIR remains

Excess bandwidth: 1M

BE gets 1 Mbps * 0.333 = 0.333 Mbps

AF1 gets 1 Mbps * 0.666 = 0.666 Mbps

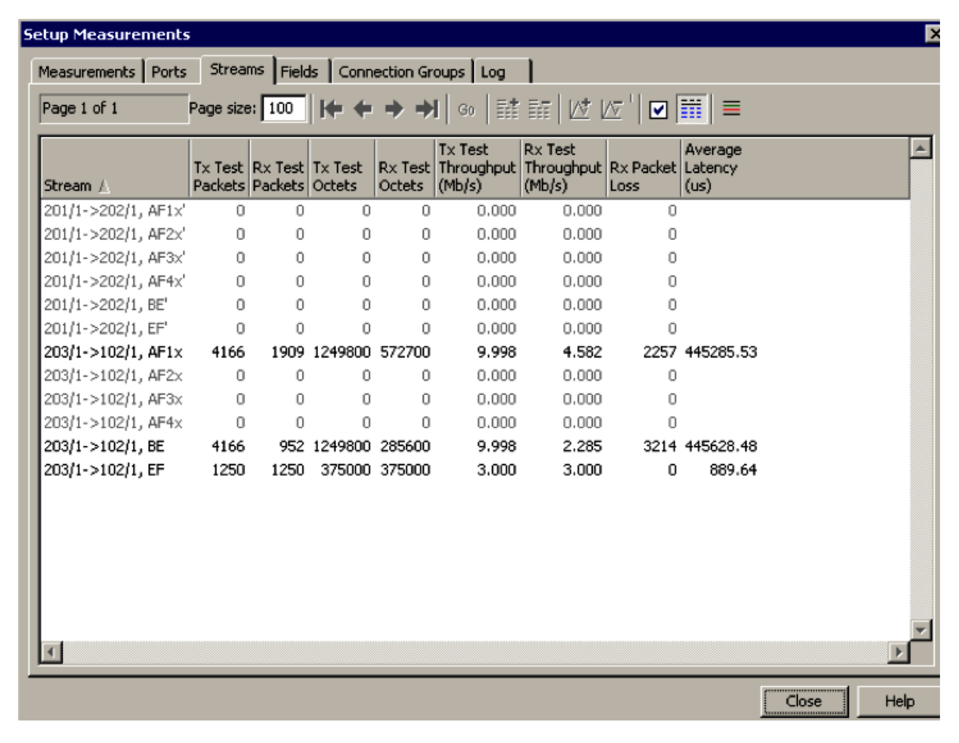
Totals:

EF: 3M

BE: 2.33 Mbps (2 Mbps + 0.333 Mbps)

AF1: 4.66 Mbps (4 M + 0.666 Mbps)

Figure 5-24 shows the measured results. While not matching exactly, they are very close and confirm the predictions for PIR mode.



Stream	Tx Test Packets	Rx Test Packets	Tx Test Octets	Rx Test Octets	Tx Test Throughput (Mb/s)	Rx Test Throughput (Mb/s)	Rx Packet Loss	Average Latency (us)
201/1->202/1, AF1x'	0	0	0	0	0.000	0.000	0	
201/1->202/1, AF2x'	0	0	0	0	0.000	0.000	0	
201/1->202/1, AF3x'	0	0	0	0	0.000	0.000	0	
201/1->202/1, AF4x'	0	0	0	0	0.000	0.000	0	
201/1->202/1, BE'	0	0	0	0	0.000	0.000	0	
201/1->202/1, EF'	0	0	0	0	0.000	0.000	0	
203/1->102/1, AF1x	4166	1909	1249800	572700	9.998	4.582	2257	445285.53
203/1->102/1, AF2x	0	0	0	0	0.000	0.000	0	
203/1->102/1, AF3x	0	0	0	0	0.000	0.000	0	
203/1->102/1, AF4x	0	0	0	0	0.000	0.000	0	
203/1->102/1, BE	4166	952	1249800	285600	9.998	2.285	3214	445628.48
203/1->102/1, EF	1250	1250	375000	375000	3.000	3.000	0	889.64

Figure 5-24. PIR Mode Measured Results.

Example 2: CIR/PIR Mode

The configuration at R1 is modified to add a 5 Mbps guaranteed rate to IFL 1 via the `tc_l3_ifl_5m` TCP:

```
{master}[edit]
jnpr@R1-RE0# show | compare
[edit class-of-service traffic-control-profiles tc_l3_ifl_5m]
-    shaping-rate 10m;
+    guaranteed-rate 5m;
[edit class-of-service interfaces xe-2/0/0 unit 1]
+      output-traffic-control-profile tc_l3_ifl_5m;

{master}[edit]
jnpr@R1-RE0# commit
re0:
. . .
```

The tc_l3_ifl_5m TCP was already in place to support a scheduler map for the queues. It had a shaping rate of 10 Mbps specified, just like the IFD level, so that the configuration would commit. An IFL-level TCP must have a G-Rate, shaping rate, or excess rate statement so the shaping rate was added to allow the commit, and use of a TCP to provide the scheduler map. Testing showed that shaping the IFL at the same speed as the IFD had no effect of measured throughput, making it a null IFL layer TCP. Without a TCP, the scheduler-map needs to be directly attached to the IFL unit.

This seems like a pretty minor change, but big things can come in small packages. Think about your answers carefully, given the new interface mode.

Pop Quiz 2 Part 1:

How will excess bandwidth be shared?

What does a transmit rate of 20% mean in this context? Will it be 2 Mbps or 1 Mbps?

What throughput do you expect for the BE queue?

Details on the scheduler and its settings are displayed:

```
NPC2(R1-RE0 vty)# sho cos scheduler-hierarchy

class-of-service EGRESS scheduler hierarchy - rates in kbps
--------------------------------------------------------------------------------
                                    shaping guarntd delaybf  excess
interface name               index    rate    rate    rate    rate      other
---------------------------- ----- ------- ------- ------- ------- -------------
xe-2/0/0                       148    10000       0       0       0
  xe-2/0/0.1                   325        0    5000       0       0
    q 0 - pri 0/0            20205        0     20%     20%      0%
    q 1 - pri 2/0            20205        0     40%     40%      0%
    q 2 - pri 3/0            20205        0     40%     40%      0%
    q 5 - pri 4/0            20205        0    3000       0      0%
  xe-2/0/0.32767               326        0    2000    2000       0
    q 0 - pri 0/1                2        0     95%     95%      0%
    q 3 - pri 0/1                2        0      5%      5%      0%
xe-2/0/1                       149        0       0       0       0
xe-2/1/0                       150        0       0       0       0
xe-2/1/1                       151        0       0       0       0
xe-2/2/0                       152        0       0       0       0
xe-2/2/1                       153        0       0       0       0
xe-2/3/0                       154        0       0       0       0
xe-2/3/1                       155        0       0       0       0

NPC2(R1-RE0 vty)# sho cos halp ifl 325
IFL type: Basic

--------------------------------------------------------------------------------
IFL name: (xe-2/0/0.1, xe-2/0/0)   (Index 325, IFD Index 148)
    QX chip id: 0
```

```
        QX chip dummy L2 index: -1
        QX chip L3 index: 3
        QX chip base Q index: 24
    Queue     State        Max       Guaranteed   Burst  Weight Priorities Drop-Rules
    Index                  rate         rate      size            G    E   Wred  Tail
    ------ ----------- ----------- ------------ ------- ------ ---------- ----------
        24  Configured    10000000      1000000  131072    125   GL   EL     4     8
        25  Configured    10000000      2000000  131072    250   GM   EL     4   136
        26  Configured    10000000      2000000  131072    250   GH   EH     4   200
        27  Configured    10000000            0  131072      1   GL   EL     0   255
        28  Configured    10000000            0  131072      1   GL   EL     0   255
        29  Configured    10000000     Disabled  131072    375   GH   EH     4   196
        30  Configured    10000000            0  131072      1   GL   EL     0   255
        31  Configured    10000000            0  131072      1   GL   EL     0   255
```

Once again, the output provided holds much gold. The biggest change is the 5 Mbps G-Rate now shown for IFL 1; this confirms a CIR/PIR mode interface. In this mode, queue bandwidth is based on G-Rate, and so all queue throughput values are now halved from the previous example, despite having the same rate percentages. The excess weights are still in the ratio of queue transmit rates but are now factored against the guaranteed rate bandwidth these rates equate to, which as noted changed from the previous PIR example. While the numbers change to reflect the BE rate moving from 2 Mbps to 1 Mbps, for example, the ratios are the same, thus the net effect is the same in that they sum to 999 and queue 1 still gets twice the excess as queue 0.

Again, think carefully before you answer.

Pop Quiz 2 Part 1 Answers:

How will excess bandwidth be shared?

Excess is still shared based on excess priority and excess weight. Q1 has its default excess priority modified such that both queue 0 and 1 have the same excess priority so they share excess based on their weight ratio, as before. However, now the excess rates are factored from the PIR, or excess region, which is 5 Mbps in this example. However, the PIR region can be reduced if needed to fulfill the CIR of GH and GM queues. As such, the CIR interface mode example brings the complexity of priority promotion and demotion into play.

What does a transmit rate of 20% mean in this context? Will it be 2 Mbps or 1 Mbps?

In CIR mode, bandwidth is based on the guaranteed rate, which is 5 Mbps in this example. Therefore, Q0's 10% rate now equates to 1 Mbps; in the PIR case, it was 2 Mbps.

What throughput do you expect for the BE queue?

5 Mbps CIR/5 Mbps PIR available

EF gets 3 Mbps of G-rate, 2 Mbps CIR/5 Mbps PIR remains

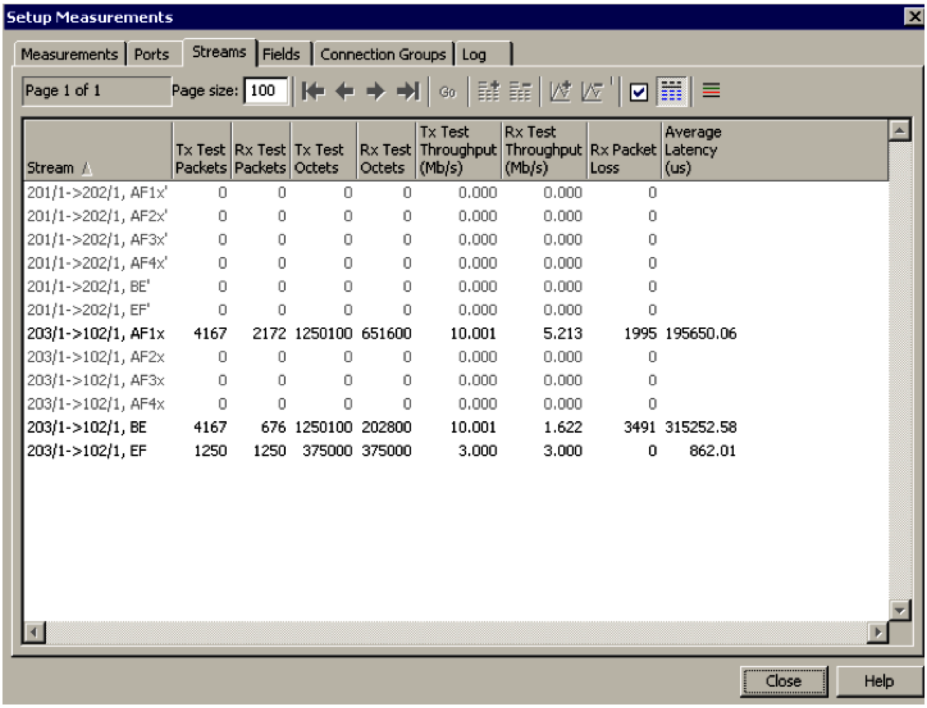
AF1 gets its 2 Mbps CIR based on GM priority/rate, 0 CIR/5 Mbps PIR remains

To avoid going into negative credit, the L3 node demotes the BE queue (was at GL) into the EL region, despite it not having sent its CIR/G-Rate. Once in GL, it no longer has a G-Rate and must contend for excess region bandwidth!

Excess bandwidth: 5 M
BE gets 5 Mbps * 0.333 = 1.665 Mbps (now at EL, demoted at L3 node)
AF1 gets 5 Mbps * 0.666 = 3.33 Mbps (now at EL, above transmit rate)
Totals:
EF: 3 Mbps (all at GH)
BE: 1.65 Mbps (all at EL)
AF1: 5.33 Mbps (2 Mbps at GM + 3.33 Mbpsat EL)

So there you go. It's all pretty straightforward, right?

Figure 5-25 shows the measured results. Again, while not an exact match, the values are very close and confirm the predictions for CIR/PIR mode.



Page 1 of 1 Page size: 100

Stream	Tx Test Packets	Rx Test Packets	Tx Test Octets	Rx Test Octets	Tx Test Throughput (Mb/s)	Rx Test Throughput (Mb/s)	Rx Packet Loss	Average Latency (us)
201/1->202/1, AF1x'	0	0	0	0	0.000	0.000	0	
201/1->202/1, AF2x'	0	0	0	0	0.000	0.000	0	
201/1->202/1, AF3x'	0	0	0	0	0.000	0.000	0	
201/1->202/1, AF4x'	0	0	0	0	0.000	0.000	0	
201/1->202/1, BE'	0	0	0	0	0.000	0.000	0	
201/1->202/1, EF'	0	0	0	0	0.000	0.000	0	
203/1->102/1, AF1x	4167	2172	1250100	651600	10.001	5.213	1995	195650.06
203/1->102/1, AF2x	0	0	0	0	0.000	0.000	0	
203/1->102/1, AF3x	0	0	0	0	0.000	0.000	0	
203/1->102/1, AF4x	0	0	0	0	0.000	0.000	0	
203/1->102/1, BE	4167	676	1250100	202800	10.001	1.622	3491	315252.58
203/1->102/1, EF	1250	1250	375000	375000	3.000	3.000	0	862.01

Figure 5-25. CIR/PIR Mode Measured Results: 5 Mbps Guaranteed Rate.

Perhaps a bit more explanation regarding the observed behavior is warranted here. If we take the EF and AF2x queues off the table, it leaves us to focus on the BE versus AF1 bandwidth sharing. These two queues are set to a 20% and 40% share of the G-Rate, yielding a 1:2 ratio, as already mentioned. Given they have the same excess priority (the default is based on their scheduling priority), they are expected to share any excess bandwidth in the same ratio. Note that if AF1x/Q1 was modified to use excess high priority it would be able to use all the excess bandwidth, in effect starving out Q0.

After EF is served, there is 2 Mbps of CIR and 5 Mbps of PIR bandwidth remaining. Queue 1, with its higher priority, gets first crack at the G-Rate bandwidth, until the queue hits its transmit rate and demotes itself; recall that at GM, this priority is not subjected to G-Rate-based demotion at a node. In this case, the AF1 queue is able to meet its transmit rate within the remaining G-Rate, but this leaves 0 G-Rate left, and given BE's demotion, and AF1 having met its transmit rate, both queues now enter excess priority.

This puts queue 0 and 1 back on equal standing, as they now both have the same EL priority, allowing them share the remaining 5 Mbps of PIR according to their weights. Table 5-13 summarizes the results of going from PIR to CIR/PIR mode while all other settings remained the same.

Table 5-13. Effects of Switching from PIR to CIR mode.

FC/Queue	Offered Load	Priority	TX/Excess rate	PIR	G-Rate: 5 Mbps
BE/0	10 M	L/EL	20%/20%	2.33 M	1.65M
AF1/1	10 M	MH/EL	40%/40%	4.66 M	5.33M
AF2/2	0 M	H	40%/40%	0	0
EF/3	3 M	SH	NA	3 M	3 M
Total				9.99 M	9.98 M

The delta between PIR and CIR mode may seem a bit surprising, given no queue parameters changed. We did mention this CoS stuff is hard to predict, right? The reason for the behavior change is that in the PIR case there was 7 Mbps of excess that was shared in a 1:2 ratio; given this was in PIR/excess range, the higher priority of queue 1 does not give it any edge over queue 0 and only its weight matters. In contrast, the CIR case had 2 Mbps of G-Rate remaining, and this allowed Q1 to use its GM priority to consume all remaining CIR, at which point the queues share the remaining 5 Mbps of excess according to their weights. The result is Q1 gets an advantage over Q0 in CIR mode that was not observed in PIR mode. A higher CIR-to-PIR ratio tends to favor higher priority queues with larger transmit weights. The PIR case was an extreme with 0 CIR, hence BE fared better there than in the CIR case where the ratio was 1:1.

It's always fun when things work to plan. Given this type of fun is not yet illegal, let's have some more!

Pop Quiz 2 Part 2:

What is the maximum rate of the AF2x queue?

Predict what will happen to BE queue throughput if the 10 Mbps AF2x flow is started.

Before you answer, it should be noted that the AF2 class is set to GH priority. This results in scheduler round-robin between AF2 and the EF queue. Also, as AF2 gets a default excess priority of EH, expect changes there. Remember, no queue is rate limited other than through the input load itself.

Also, there is no configuration change. This is a 100% data-driven change in behavior based on the absence or presence of AF2x traffic, the latter bringing it higher priority, both for CIR and excess, into the fray.

Refer to the previous scheduler output for specifics as needed.

Pop Quiz 2 Part 2 Answers:

What is the maximum rate of the AF2x queue?

You might be tempted to think that without contention from other queues, and lacking any form of rate limit or shaping rate, the AF1 queue can send up to IFL-shaped speed, or 10 Mbps in this example. The presence of EF at the same priority will limit it to no more than 7 Mbps, however. While reasonable, this is not the case.

Though perhaps not expected, the presence of AF1 at GM priority has an impact on AF2's maximum rate. This is because L2/L3 schedulers must honor GH and GM CIRs, a behavior that stems from the node not being able to demote these priorities. The result is that the scheduler ends up going into 2 Mbps negative G-Rate credits to please both, which effectively adds 2 Mbps to the IFL's G-Rate. The extra bandwidth comes from the excess region, effectively taking it from AF2 where it will otherwise dominate all excess bandwidth given its higher priority. Thus AF2 is expected to get no more than 5 Mbps of bandwidth when both EF and AF1 are flowing.

Predict what will happen to BE queue throughput if the 10 Mbps AF2x flow is started.

A lot changes given AF2's GH/EH priority. The higher excess priority and lack of rate limiting means this queue can dominate all PIR/excess region bandwidth. The BE queue's GL priority makes it eligible for demotion at the L3 scheduler node, making its G-Rate no longer guaranteed. It seems that the venerable BE queue may soon feel rather slighted by the scheduler as it enters a famine state.

5 Mbps CIR/5 Mbps PIR available

EF gets 3 Mbps, 2 Mbps CIR/5 Mbps PIR remains

AF2 gets 2 Mbps based on GH priority/rate, 0 CIR/5 Mbps PIR remains

AF1 gets 2 Mbps based on GM priority/rate, –2 M CIR/3 M PIR remains. L2 and L3 schedulers have to honor GH/GM guaranteed rates! BE at GL is demoted into excess region at EL

Excess bandwidth: 3 M

AF2 gets all 3 Mbps (now at EH, starves both AF1 and BE for excess/PIR bandwidth)

Totals:

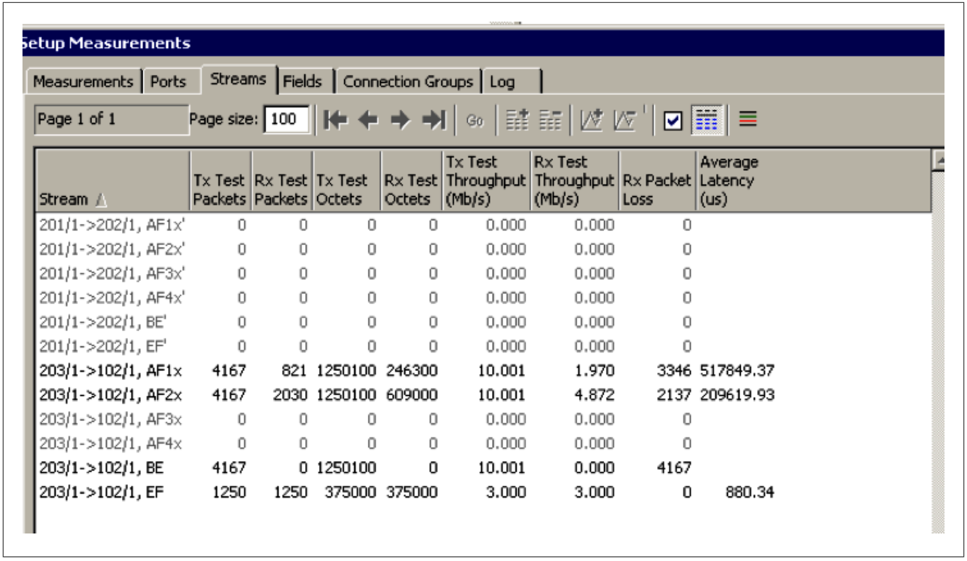
EF: 3 Mbps (all at GH)

BE: 0 Mbps (demoted to EL at L3, starved in PIR region by AF2)

AF1: 2 Mbps (all at GM, starved in PIR region)

AF2: 5 Mbps (2 Mbps at GH + all 3 Mbps of PIR region)

Figure 5-26 shows the measured results. Again, this is close enough to confirm predictions of Trio behavior are possible, though at first this may not seem the case.



Stream	Tx Test Packets	Rx Test Packets	Tx Test Octets	Rx Test Octets	Tx Test Throughput (Mb/s)	Rx Test Throughput (Mb/s)	Rx Packet Loss	Average Latency (us)
201/1->202/1, AF1x'	0	0	0	0	0.000	0.000	0	
201/1->202/1, AF2x'	0	0	0	0	0.000	0.000	0	
201/1->202/1, AF3x'	0	0	0	0	0.000	0.000	0	
201/1->202/1, AF4x'	0	0	0	0	0.000	0.000	0	
201/1->202/1, BE'	0	0	0	0	0.000	0.000	0	
201/1->202/1, EF'	0	0	0	0	0.000	0.000	0	
203/1->102/1, AF1x	4167	821	1250100	246300	10.001	1.970	3346	517849.37
203/1->102/1, AF2x	4167	2030	1250100	609000	10.001	4.872	2137	209619.93
203/1->102/1, AF3x	0	0	0	0	0.000	0.000	0	
203/1->102/1, AF4x	0	0	0	0	0.000	0.000	0	
203/1->102/1, BE	4167	0	1250100	0	10.001	0.000	4167	
203/1->102/1, EF	1250	1250	375000	375000	3.000	3.000	0	880.34

Figure 5-26. CIR/PIR Mode with AF2x.

Example 3: Make a Small, "Wafer-thin" Configuration Change

Things are returned to the initial CIR interface mode state; the AF2x traffic is again disabled. Recall that previous displays confirmed the excess rate weighting was based on the ratio of queue transmit rate. The values are shown again to refresh:

```
NPC2(R1-RE0 vty)# sho cos halp ifl 325
IFL type: Basic

--------------------------------------------------------------------------------
IFL name: (xe-2/0/0.1, xe-2/0/0)   (Index 325, IFD Index 148)
    QX chip id: 0
    QX chip dummy L2 index: -1
    QX chip L3 index: 7
    QX chip base Q index: 56
Queue     State         Max       Guaranteed   Burst  Weight Priorities Drop-Rules
Index                   rate         rate      size            G    E   Wred  Tail
------ ----------- ----------- ------------ ------- ------ ---------- ----------
    56  Configured    10000000      1000000  131072    125   GL    EL     4      8
    57  Configured    10000000      2000000  131072    250   GM    EL     4    136
    58  Configured    10000000      2000000  131072    250   GH    EH     4    200
    59  Configured    10000000            0  131072      1   GL    EL     0    255
    60  Configured    10000000            0  131072       1    GL    EL     0    255
    61  Configured    10000000     Disabled  131072     375    GH    EH     4    196
    62  Configured    10000000            0  131072       1    GL    EL     0    255
    63  Configured    10000000            0  131072       1    GL    EL     0    255
```

In this example, your goal is to simply add an explicit `excess-rate proportion` configuration to the BE queue that matches the current default value of 125; given it's the same value being explicitly set, this seems like a null operation as far as CoS behavior changes:

```
{master}[edit]
jnpr@R1-RE0# set class-of-service schedulers sched_be_20 excess-rate proportion 125

{master}[edit]
jnpr@R1-RE0# show | compare
[edit class-of-service schedulers sched_be_20]
+    excess-rate proportion 125;

{master}[edit]
jnpr@R1-RE0# commit
re0:
```

Now, this is most definitely a minor change, right?

Pop Quiz 3:

How will excess bandwidth be shared?

What throughput do you expect for the BE queue?

Again, think about your answers carefully and factor them against the new scheduler setting outputs shown:

```
NPC2(R1-RE0 vty)# sho cos scheduler-hierarchy

class-of-service EGRESS scheduler hierarchy - rates in kbps
---------------------------------------------------------------------------------
                                        shaping guarntd delaybf  excess
interface name                  index     rate    rate    rate    rate      other
---------------------------- -----  ------- ------- ------- ------- -------------
xe-2/0/0                          148    10000       0       0       0
  xe-2/0/0.1                      325        0    5000       0       0
    q 0 - pri 0/0               20205        0     20%     20%     125
    q 1 - pri 2/0               20205        0     40%     40%      0%
    q 2 - pri 3/0               20205        0     40%     40%      0%
    q 5 - pri 4/0               20205        0    3000       0      0%
  xe-2/0/0.32767                  326        0    2000    2000       0
    q 0 - pri 0/1                   2        0     95%     95%      0%
    q 3 - pri 0/1                   2        0      5%      5%      0%
xe-2/0/1                          149        0       0       0       0
xe-2/1/0                          150        0       0       0       0
xe-2/1/1                          151        0       0       0       0
xe-2/2/0                          152        0       0       0       0
xe-2/2/1                          153        0       0       0       0
xe-2/3/0                          154        0       0       0       0
xe-2/3/1                          155        0       0       0       0
```

```
NPC2(R1-RE0 vty)# sho cos halp ifl 325
IFL type: Basic

--------------------------------------------------------------------------------
IFL name: (xe-2/0/0.1, xe-2/0/0)   (Index 325, IFD Index 148)
    QX chip id: 0
    QX chip dummy L2 index: -1
    QX chip L3 index: 3
    QX chip base Q index: 24
Queue    State         Max       Guaranteed   Burst  Weight Priorities Drop-Rules
Index                  rate         rate      size            G    E   Wred  Tail
------ ----------- ----------- ------------ ------- ------ ---------- ----------
    24  Configured    10000000      1000000  131072   1000   GL   EL     4     8
    25  Configured    10000000      2000000  131072      1   GM   EL     4   136
    26  Configured    10000000      2000000  131072      1   GH   EH     4   200
    27  Configured    10000000            0  131072      1   GL   EL     0   255
    28  Configured    10000000            0  131072      1   GL   EL     0   255
    29  Configured    10000000     Disabled  131072      1   GH   EH     4   196
    30  Configured    10000000            0  131072      1   GL   EL     0   255
    31  Configured    10000000            0  131072      1   GL   EL     0   255
```

In case it was missed, the key change here is the explicit setting places the interface into explicit excess rate mode. In this mode, any queue that lacks an excess rate setting gets a default exceeds rate of 0, yielding them a weighting of 1. With AF2x again out of the picture due to no input stimulus, it's expected that BE will now dominate the excess bandwidth region.

Pop Quiz 3 Answers:

How will excess bandwidth be shared?

There is no change in behavior here. Excess is shared based on priority and excess weight. With AF2 out of contention, the BE queues large excess weight will allow it to dominate the excess region.

What throughput do you expect for the BE queue?

5 Mbps CIR/5 Mbps PIR available

EF gets 3 Mbps, 2 Mbps CIR/5 Mbps PIR remains

AF1 gets 2 Mbps based on MH priority/rate, 0 CIR/7 Mbps PIR remains, node enters PIR region, BE at GL is demoted. AF1 enters excess having reached its transmit rate

Excess bandwidth: 5 Mbps queue 0/1 now at 1,000:1 ratio

BE gets 5 Mbps * 0.999 = 4.995 Mbps (no G-rate, all at EL)

AF1 gets 5 Mbps * 0.001 = 0.005 Mbps (no G-rate, now at EL)

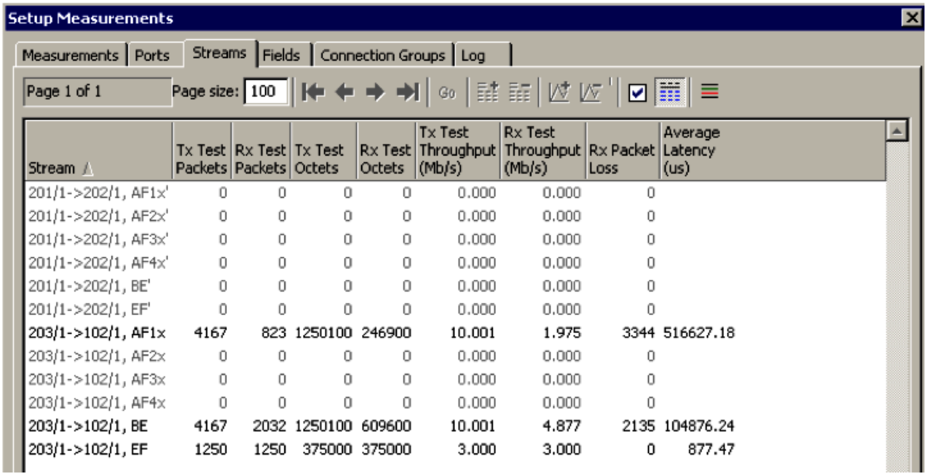
Totals:

EF: 3 Mbps (all at GH)

BE: 4.99 Mbps (all at EL)

AF1: 2 Mbps (2 Mbps at GM + 0.005 Mbps at EL)

Figure 5-27 shows the measured results. While not matching exactly, they are very close and confirm the predictions for the *slightly modified* CIR/PIR mode experiment.



Stream	Tx Test Packets	Rx Test Packets	Tx Test Octets	Rx Test Octets	Tx Test Throughput (Mb/s)	Rx Test Throughput (Mb/s)	Rx Packet Loss	Average Latency (us)
201/1->202/1, AF1x'	0	0	0	0	0.000	0.000	0	
201/1->202/1, AF2x'	0	0	0	0	0.000	0.000	0	
201/1->202/1, AF3x'	0	0	0	0	0.000	0.000	0	
201/1->202/1, AF4x'	0	0	0	0	0.000	0.000	0	
201/1->202/1, BE'	0	0	0	0	0.000	0.000	0	
201/1->202/1, EF'	0	0	0	0	0.000	0.000	0	
203/1->102/1, AF1x	4167	823	1250100	246900	10.001	1.975	3344	516627.18
203/1->102/1, AF2x	0	0	0	0	0.000	0.000	0	
203/1->102/1, AF3x	0	0	0	0	0.000	0.000	0	
203/1->102/1, AF4x	0	0	0	0	0.000	0.000	0	
203/1->102/1, BE	4167	2032	1250100	609600	10.001	4.877	2135	104876.24
203/1->102/1, EF	1250	1250	375000	375000	3.000	3.000	0	877.47

Figure 5-27. CIR/PIR Mode: Effects of Explicit Excess Priority on One Queue.

Predicting Queue Throughput Summary

The examples in this section were designed to help the reader put many abstract concepts and facts regarding Trio H-CoS scheduling behavior to a practical test. Doing so helps explain why predicting CoS behavior can be difficult and helps stress the need to test and model CoS changes before simply putting them into production.

The next section builds on these points by demonstrating Trio CoS as part of an end-to-end CoS solution.

CoS Lab

OK, after all that talk it's time to get down to CoS business. In this section, you enable your Trio-based MX network for L2 and L3 CoS, and then verify that all has gone to plan. Figure 5-28 shows the CoS test topology.

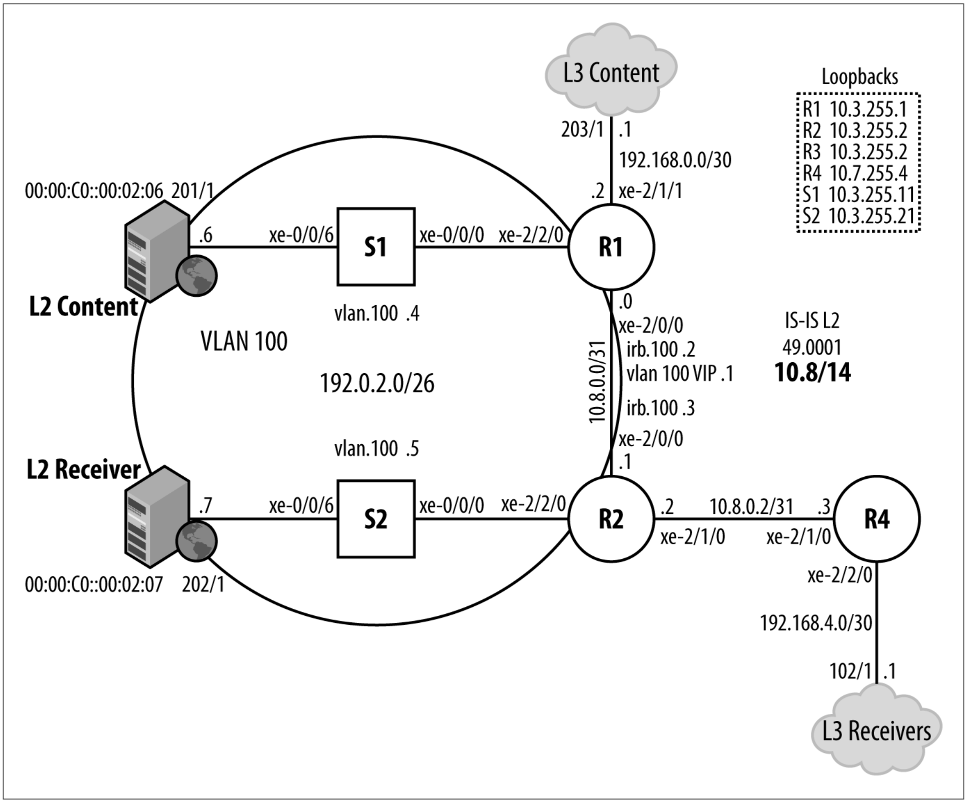


Figure 5-28. The Trio CoS Test Topology.

The test topology does not use AE interfaces given that H-CoS support for IFL-Sets on AE is not present in 11.4R1. A single 10 Gbps link is used between the switches and routers. R1 and R2 both serve a Layer 2 domain with VLAN 100 the area of focus. The VLAN is assigned logical IP subnet (LIS) 192.0.2.0/25. R1's irb.100 interface is assigned 192.0.2.2, while R2 is given 192.0.2.3. The VRRP VIP, owned by R1 when operational, is 192.0.2.1.

The redundant links in the L2 domain have been eliminated to constrain traffic to the single path between the L2 source and receiver to help keep the focus on CoS. IS-IS level 2 is enabled on the core backbone links between R1, R2, and R4. Passive mode is configured on the Layer 3 source and receiver ports to ensure reachability to the associated 192.168.x.x/30 subnetworks.

The initial goal is to configure and verify IP CoS is working for both L2 switched and L3 routed traffic, with the focus on R1's configuration and operation.

Configure Unidirectional CoS

Virtually all IP networks are duplex in nature, which is to say traffic is both sent and received. A CoS design tends to be symmetric as well, providing the same sets of classification, rewrite, and scheduling behavior in both directions. This is not a mandate, however. When learning CoS in a test network, the authors believe it makes sense to focus on a single direction. Once you have the steps down, and things are working to your satisfaction, it's relatively trivial to then extend the CoS design into the other direction.

Remember, CoS only matters when things get congested. If links are less than 80% utilized, there is no real queuing; hence, things just work. Thus, in our test network where we have full control over traffic volumes as well as who sends and who receives, it's safe to leave default CoS or partial CoS in the "receive" direction, again knowing there is no congestion in this direction and therefore no chance for lost ACKs that might skew results if, for example, one was conducting stateful TCP-based throughput testing (which we are not, as all test traffic is IP-based with no TCP or UDP transport).

As shown in Figure 5-28, traffic in this lab moves from top to bottom. The L2 bridged traffic originates at port 201/1, flows via S1, R1, R2, and then S2, to arrive at receiver port 202/1. In like fashion, the L3 routed traffic originates at port 203/1, transits R1, R2, and then R4, to arrive at port 102/1. All links are either L2 or L3, with the exception of the R1-R2 link, which has two units: 0 for bridged and 1 for routed IP. The MAC addresses for the two router tester ports in the L2 domain are documented, as is the IP addressing assignments for all L3 interfaces.

Before starting on the CoS, a quick spot check of the configuration and some operational checks are performed:

```
{master}[edit]
jnpr@R1-RE0# show interfaces xe-2/0/0
hierarchical-scheduler;
vlan-tagging;
unit 0 {
    family bridge {
        interface-mode trunk;
        vlan-id-list 1-999;
    }
}
unit 1 {
    vlan-id 1000;
    family inet {
        address 10.8.0.0/31;
    }
    family iso;
}

{master}[edit]
jnpr@R1-RE0# run show isis adjacency
Interface             System         L State        Hold (secs) SNPA
xe-2/0/0.1            R2-RE0         2  Up                   21
```

The display confirms the xe-2/0/0 interface's L2 and L3 configuration, and that the IS-IS adjacency to R2 is operational. VRRP and STP state is checked:

```
{master}[edit]
jnpr@R1-RE0# run show vrrp
Interface     State       Group   VR state VR Mode   Timer    Type   Address
irb.100       up              0   master   Active      A  0.072 lcl    192.0.2.2
                                                                vip    192.0.2.1
irb.200       up              1   master   Active      A  0.306 lcl    192.0.2.66
                                                                vip    192.0.2.65

{master}[edit]
jnpr@R1-RE0# run show spanning-tree bridge vlan-id 100
STP bridge parameters
Routing instance name               : GLOBAL
Enabled protocol                    : RSTP

STP bridge parameters for VLAN 100
  Root ID                           : 4196.00:1f:12:b8:8f:d0
  Hello time                        : 2 seconds
  Maximum age                       : 20 seconds
  Forward delay                     : 15 seconds
  Message age                       : 0
  Number of topology changes        : 6
  Time since last topology change   : 243505 seconds
  Local parameters
    Bridge ID                       : 4196.00:1f:12:b8:8f:d0
    Extended system ID              : 100
```

The output confirms that R1 "is like a boss," as least from the perspective of VLAN 100 and VRRP; it's the root of the STP and the current VIP master. Routes to all loopback interfaces and to the L3 content ports is also verified:

```
{master}[edit]
jnpr@R1-RE0# run show route protocol isis

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

10.3.255.2/32      *[IS-IS/18] 2d 19:34:41, metric 10
> to 10.8.0.1 via xe-2/0/0.1
10.7.255.4/32      *[IS-IS/18] 2d 19:34:41, metric 20
> to 10.8.0.1 via xe-2/0/0.1
10.8.0.2/31        *[IS-IS/18] 2d 19:34:41, metric 20
> to 10.8.0.1 via xe-2/0/0.1
192.0.2.192/26     *[IS-IS/18] 2d 19:34:41, metric 83
> to 10.8.0.1 via xe-2/0/0.1
192.168.4.0/30     *[IS-IS/18] 2d 19:34:41, metric 30
> to 10.8.0.1 via xe-2/0/0.1

iso.0: 1 destinations, 1 routes (1 active, 0 holddown, 0 hidden)
```

And, connectivity is confirmed between the loopbacks of R1 and R4:

```
{master}[edit]
jnpr@R1-RE0# run ping 10.7.255.4 source 10.3.255.1
```

```
PING 10.7.255.4 (10.7.255.4): 56 data bytes
64 bytes from 10.7.255.4: icmp_seq=0 ttl=63 time=0.726 ms
64 bytes from 10.7.255.4: icmp_seq=1 ttl=63 time=0.636 ms
64 bytes from 10.7.255.4: icmp_seq=2 ttl=63 time=0.620 ms
^C
--- 10.7.255.4 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.620/0.661/0.726/0.047 ms

{master}[edit]
jnpr@R1-RE0# run traceroute 10.7.255.4 no-resolve
traceroute to 10.7.255.4 (10.7.255.4), 30 hops max, 40 byte packets
 1  10.8.0.1  0.459 ms  0.342 ms  0.331 ms
 2  10.7.255.4  2.324 ms  0.512 ms  0.453 ms
```

Note that for R1, the 192.168.0/30 subnet is a direct connection, and so is not learned through IS-IS. The passive setting is verified at R1:

```
{master}[edit]
jnpr@R1-RE0# run show isis interface
IS-IS interface database:
Interface             L CirID Level 1 DR        Level 2 DR        L1/L2 Metric
irb.100               0   0x1 Passive           Passive               100/100
lo0.0                 0   0x1 Passive           Passive                 0/0
xe-2/0/0.1            2   0x1 Disabled          Point to Point         10/10
xe-2/1/1.0            0   0x1 Passive           Passive                10/10
```

Also of note, given the last display, is that IS-IS is also set for passive option on the IRB interfaces. As with the L3 content ports, this setting ensures that the related Layer 2 network IP subnet is advertised into the Layer 3 domain to accommodate routing traffic into the bridged network. On the bridged side, a default gateway/default route is used to route inter-VLAN traffic by directing it to the VLANs default gateway, which here is the VIP address 192.2.0.1, currently owned by R1. Connectivity from L2 into L3, via the IRB at R1, is verified at S1:

```
{master:0}[edit]
jnpr@S1-RE0# show routing-options
static {
    route 0.0.0.0/0 next-hop 192.0.2.1;
}

{master:0}[edit]
jnpr@S1-RE0#

{master:0}[edit]
jnpr@S1-RE0# run traceroute 192.168.4.2 no-resolve
traceroute to 192.168.4.2 (192.168.4.2), 30 hops max, 40 byte packets
 1  192.0.2.2  1.053 ms  0.679 ms  6.107 ms
 2  10.8.0.1  2.256 ms  0.666 ms  0.614 ms
 3  192.168.4.2  2.181 ms  0.862 ms  3.691 ms
```

With the baseline network's operation verified, we move into the realm of Trio CoS.

Establish a CoS Baseline

CoS configurations tend to be long and repetitive, in that the majority of the CoS settings found in one node are likely to be found in the next. One hallmark of a successful CoS design is, after all, consistent and predictable handling of traffic on a node-by-node basis; a feat that is appreciably more complicated when all routers have random of differing configurations. Getting a baseline CoS design up and running can seem overwhelming, but as with all complex subjects, if taken one chunk at a time, the individual pieces are easily managed. Establishing the CoS baseline for your network is the hard part; after that, it's mostly just tuning and tweaking to accommodate new services or to fine tune operation.

Keep in mind the IP DiffServ model and basic Junos CoS processing are detailed in the *Junos Enterprise Routing* book if a review of the basics is desired. It's clear you must consider the following questions in order to establish a CoS baseline for your network.

What protocols are transported?
: Clearly, the answer here determines what type of classifiers are needed, how packets are rewritten, and ultimately how many sets of classification or rewrite rules end up applied to an interface. The good news is you can deploy CoS starting with one protocol and then add support for additional protocols incrementally. This approach greatly lessens the daunting factor of trying to deploy CoS at multiple levels, for multiple protocols, at the same time, and allows you to leverage existing work, such as scheduler definitions, which are just as applicable to IPv6 as they are to bridged traffic; schedulers are protocol agnostic, after all.

How many forwarding classes (FCs)/queues?
: Most networks need at least three, with the trend being to use as many as eight. In Junos, an FC generally maps on a 1-to-1 basis to a queue, but this is not always the case. As queue handling is ultimately where the CoS rubber meets the road, so to speak, best practice dictates you should only have as many FCs as you can uniquely provide CoS handling for, which means FCs should equal queues on a 1:1 basis.

Classification type, MF or BA?
: Generally the answer here is both, as MF classification using a firewall filter tends to be performed at the edges, while BA classification is done in the core. Even so, there may be multiple options as to what type of BA to use, for example IP DSCP versus IP precedence. In many cases, IP is encapsulated into VPLS or MPLS, which means the appropriate L2/MPLS EXP classifiers and rewrite rules are needed on P-routers.

  As an example, consider an L3VPN environment using MPLS. Here, it's typical that PE routers use IP-based classification at ingress and then rewrite both the IP and MPLS headers at egress. P-routers along the path use the tunnel encapsulation for CoS classification. The egress router normally receives a single label packet due to Penultimate Hop Popping (PHP), where the second-to-last hop router pops, rather than swaps, the outer transport label, resulting in receipt of a packet with a single VRF label at the egress node. In these cases, Trio PFEs can perform IP-level classification as an egress node, such that the egress filters and rewrites rules for the traffic heading to the remote CE are based on the IP layer.

Per forwarding class (queue) QoS parameters

This is where most of the brain work comes in. The cumulative effects of these parameters provide *the CoS* handling for a given FC. By assigning different values to these parameters, you legitimize your network's CoS, which is to say it's here that you actually instantiate differing levels of service; if you assign the same parameters to all eight FCs, it can be said that your network has no CoS, as in the end all traffic is treated the same.

QoS parameters for queues

You assign queue parameters through a scheduler definition and then apply one or more schedulers to an interface with a scheduler map. Scheduler parameters for queues include:

Scheduling priority

Transmit rate

Excess rate/priority

Delay buffer size

Drop profiles

Shaping

The scheduling mode

Options here are port-level, per unit, or hierarchical. Many incorrectly believe that the choice of scheduler mode must be deployed consistently on a networkwide basis. In fact, this is a choice that can be made on a per port basis and is generally dictated by your need for granular CoS at scale versus simply providing CoS differentiation among FCs.

Any interface with a single logical unit defined is a prime candidate for per port mode, which conveniently is supported on all Trio MPCs. If your interface has multiple units, you might consider using per unit mode if you need to handle the queues from one unit differently than another. If all units support the same nature of traffic, and you plan on scheduling all queues in the same fashion, then clearly per port is a better choice; the overall effect is the same and the line card is cheaper, as fine-grained queuing supported is not required.

In contrast, hierarchical mode is the only practical option if you are in the subscriber aggregation businesses and the plan is to deploy high-speed interfaces with literally thousands of IFLs, over which you need to offer multiple levels of services, for example business class versus residential Internet, or triple play (Internet/voice/video).

To help flesh out the requirements to narrow down the otherwise huge ream of possibilities, consider these CoS design goals for the CoS lab:

Provide CoS for both native Layer 2 bridged traffic, routed traffic, and traffic that flows between bridged and routed domains

Support eight FCs and eight queues

Trust L3 DSCP markings, use BA classification and rewrite to preserve them

Use MF classification for bridged traffic to support DiffServ with four AFx classes along with EF, NC, and BE

Provide isolation between Layer 2 bridged and Layer 3 routed traffic; excessive levels of one should not impact the CoS of the other

The scheduling goals for the CoS lab are:

Provide a real-time LLQ for voice and video with 25 milliseconds of delay per node and 30% of interface bandwidth; ensure this queue cannot cause starvation for other classes

Ensure that network control queue has priority over all non-EF queues, again ensure no starvation for other classes

Support all four Assured-Forwarding (AF) classes according to IP DiffServ

Provide a penalty box queue for bad customers or traffic

Share excess bandwidth among all eligible FCs

Given the requirements, the plan is to use eight FCs/queues, DSCP-based BA for IPv4, and MF classification for bridged traffic. As noted, intra-VLAN L2 traffic and routed L3 traffic are both natively transported over the R1-R2 link using different units/IFLs. MF classification is performed on the MX for the L2 traffic for several reasons. Use of a MF classifier ties in well with Chapter 3, and its use overcomes the inherit limitations of IEEE 802.1p-based classification, which, like IP precedence, supports only eight combinations (their respective bit fields are each 3 bits in length). In contrast, a 6-bit DSCP codes up to 64 combinations and full DiffServ support needs more than eight code points.

The decision is made to provide different levels of CoS for the two traffic types, which indicates the need for per unit scheduling, or a separate interface is needed between R1 and R2 so it can be dedicated to L2 traffic in order to meet the stated traffic isolation requirement. H-CoS is overkill in this application but could also work.

As noted previously, per unit scheduling not only allows for different CoS profiles, but also helps ensure a degree of separation between the bridged and routed traffic so that abnormal traffic levels at one layer doesn't necessarily impact the operation of the other. For example, even with storm control enabled, if a loop forms in the bridged network, significant bandwidth can be consumed on the R1-R2 link, bandwidth that is far in excess of normal bridged loads. If port mode CoS is used, the shared set of queues would be disproportionally filled with L2 traffic, resulting poor L3 performance. While ingress policing could be used to help mitigate these concerns in such a design, the routers in this lab have dense queuing-capable MPCs, so per unit and H-CoS are supported and you might as well use what you pay for.

Baseline Configuration. With these criteria in mind, the baseline CoS settings are displayed. Things start with the FC definitions, which is then backed up via operational mode CLI commands:

```
{master}[edit]
jnpr@R1-RE0# show class-of-service forwarding-classes
class be queue-num 0 policing-priority normal;
class af1x queue-num 1 policing-priority normal;
class af2x queue-num 2 policing-priority normal;
class nc queue-num 3 policing-priority normal;
class af4x queue-num 4 policing-priority normal;
class ef queue-num 5 priority high policing-priority premium;
class af3x queue-num 6 policing-priority normal;
class null queue-num 7 policing-priority normal;

{master}[edit]
jnpr@R1-RE0# run show class-of-service forwarding-class
Forwarding  ID   Queue  Restricted  Fabric     Policing    SPU
class                    queue        priority   priority     priority
  be        0    0        0           low        normal       low
  af1x      1    1        1           low        normal       low
  af2x      2    2        2           low        normal       low
  nc        3    3        3           low        normal       low
  af4x      4    4        0           low        normal       low
  ef        5    5        1           high       premium      low
  af3x      6    6        2           low        normal       low
  null      7    7        3           low        normal       low
```

The output confirms eight DiffServ-based FCs are defined. Note that NC has been left in queue 3, a good practice, and that the EF class has been set to a high switch fabric priority, to provide it preferential treatment in the event of congestion across the fabric; preclassification ensures that NC is always sent over the fabric at high priority. The policing priority value of premium or normal is used for aggregate policers, as described in Chapter 3. While aggregate policers are not planned in the current CoS scenario, they can always be added later, and having an explicit configuration in place helps ensure things work correctly the first time. You may want to note the queue number to FC mappings, as some of the subsequent displays list only the queue and its internal index number, in which case they are always listed from 0 to 7. Trio platforms are inherently eight-queue capably so no additional configuration is needed to use eight queues per IFL.

Next, the DSCP classifier is displayed. This BA classifier is applied to all L3 interfaces:

```
{master}[edit]
jnpr@R1-RE0# sh class-of-service classifiers dscp dscp_diffserv
forwarding-class ef {
    loss-priority low code-points ef;
}
forwarding-class af4x {
    loss-priority low code-points af41;
    loss-priority high code-points [ af42 af43 ];
}
forwarding-class af3x {
    loss-priority high code-points [ af32 af33 ];
    loss-priority low code-points af31;
}
forwarding-class af2x {
    loss-priority low code-points af21;
    loss-priority high code-points [ af22 af23 ];
}
forwarding-class af1x {
    loss-priority low code-points af11;
    loss-priority high code-points [ af12 af13 ];
}
forwarding-class nc {
    loss-priority low code-points [ cs6 cs7 ];
    loss-priority high code-points [ cs1 cs2 cs3 cs4 cs5 ];
}
forwarding-class be {
    loss-priority low code-points [ 000000 000001 000010 000011 000100 000101 000110
      000111 001001 001011 001101 001111 010001 010011 010101 010111 011001 011011
      011101 011111 100001 100011 100101 100111 101001 101010 101011 101100 101101
      101111 110001 110010 110011 110100 110101 110110 110111 111001 111010 111011
      111100 111101 111110 111111 ];
}

{master}[edit]
jnpr@R1-RE0# run show class-of-service interface xe-2/1/1 | match class
    Classifier              dscp_diffserv          dscp                    23080
```

And now the IEEE 802.1p classifier, which is based on 3-bit Priority Code Point (PCP) field found in the VLAN tag. Clearly, with 3 bits available, only 8 combinations are possible; this results in a loss of granularity when compared to the 64 combinations offered by the 6-bit DSCP field. Perhaps it's close enough and you are happy with partial DiffServ support. If not, the power of Trio chipsets comes to the rescue as MF classification can be used to override any BA classifications and are a viable option where the extra granularity is needed. As noted, this example ultimately uses MF classification based on IP DSCP to demonstrate this very concept, but still applies the L2 BA classifier as part of best practice design. Just another layer of consistency, and one less chance for a packet being unclassified, and thereby given BE treatment.

```
{master}[edit]
jnpr@R1-RE0# show class-of-service classifiers ieee-802.1 ieee_classify
forwarding-class be {
    loss-priority low code-points 000;
    loss-priority high code-points 111;
}
forwarding-class af1x {
    loss-priority low code-points 001;
}
forwarding-class af2x {
    loss-priority low code-points 010;
}
forwarding-class nc {
    loss-priority low code-points 011;
}
```

```
forwarding-class af4x {
    loss-priority low code-points 100;
}
forwarding-class ef {
    loss-priority low code-points 101;
}
forwarding-class af3x {
    loss-priority low code-points 110;
}

{master}[edit]
jnpr@R1-RE0# run show class-of-service interface xe-2/2/0 | match class
    Classifier              ieee_classify         ieee8021p               22868
```

As you would expect, the R1-R2 link has both the L2 and L3 BA classifiers in effect:

```
{master}[edit]
jnpr@R1-RE0# run show class-of-service interface xe-2/0/0
Physical interface: xe-2/0/0, Index: 148
Queues supported: 8, Queues in use: 8
Total non-default queues created: 0
  Scheduler map: <default>, Index: 2
  Congestion-notification: Disabled

  Logical interface: xe-2/0/0.0, Index: 332
    Object                  Name                  Type                    Index
    Rewrite                 ieee_rewrite           ieee8021p (outer)       16962
    Classifier              ieee_classify         ieee8021p               22868

  Logical interface: xe-2/0/0.1, Index: 333
    Object                  Name                  Type                    Index
    Rewrite                 dscp_diffserv         dscp                    23080
    Classifier              dscp_diffserv         dscp                    23080

  Logical interface: xe-2/0/0.32767, Index: 334
```

As noted previously, the automatically generated internal unit 32767, along with its automatically generated scheduler, is used to handle LACP control protocol traffic sent over the VLAN-tagged interface. This output also shows that both DSCP and IEEE 802.1p-based rewrite rules are in effect, again on a logical unit basis and according to the protocol family that is configured on that unit. To save space, the rewrite rules are not displayed. You may assume they are consistent with the classifiers shown above, and that all interfaces have both a BA classifier as well as a set of rewrite rules in effect, as per the following:

```
{master}[edit]
jnpr@R1-RE0# show class-of-service interfaces
xe-2/0/0 {
    unit 0 {
        classifiers {
            ieee-802.1 ieee_classify;
        }
        rewrite-rules {
            ieee-802.1 ieee_rewrite;
        }
    }
    unit 1 {
        classifiers {
            dscp dscp_diffserv;
        }
        rewrite-rules {
            dscp dscp_diffserv;
        }
    }
}
xe-2/1/1 {
    unit 0 {
        classifiers {
            dscp dscp_diffserv;
        }
        rewrite-rules {
            dscp dscp_diffserv;
        }
    }
}
xe-2/2/0 {
    unit 0 {
        classifiers {
            ieee-802.1 ieee_classify;
        }
        rewrite-rules {
            ieee-802.1 ieee_rewrite;
        }
    }
}
```

Because we are in the area of classification, the MF classifier used at R1 for DSCP classification in the context of family bridge traffic received from the Layer 2 source at S1 is displayed. Note the filter is applied in the input direction to catch received traffic:

```
{master}[edit]
jnpr@R1-RE0# show interfaces xe-2/2/0
unit 0 {
    family bridge {
        filter {
            input l2_mf_classify;
        }
        interface-mode trunk;
        vlan-id-list 1-999;
    }
}

{master}[edit]
jnpr@R1-RE0# show firewall family bridge
filter l2_mf_classify {
    term ef {
        from {
            ether-type ipv4;
            dscp ef;
        }
```

```
            then {
                count ef;
                forwarding-class ef;
                accept;
            }
        }
        term af11 {
            from {
                ether-type ipv4;
                dscp af11;
            }
            then {
                count af11;
                forwarding-class af1x;
                accept;
            }
        }
        term af1x {
            from {
                ether-type ipv4;
                dscp [ af12 af13 ];
            }
            then {
                count af1x;
                loss-priority high;
                forwarding-class af1x;
                accept;
            }
        }
        term af21 {
            from {
                ether-type ipv4;
                dscp af21;
            }
            then {
                count af21;
                forwarding-class af2x;
                accept;
            }
        }
        term af2x {
            from {
                ether-type ipv4;
                dscp [ af22 af23 ];
            }
            then {
                count af2x;
                loss-priority high;
                forwarding-class af2x;
                accept;
            }
        }
        term af31 {
            from {
                ether-type ipv4;
                dscp af31;
            }
            then {
                count af31;
                forwarding-class af3x;
                accept;
            }
        }
        term af3x {
            from {
                ether-type ipv4;
                dscp [ af32 af33 ];
            }
            then {
                count af3x;
                loss-priority high;
                forwarding-class af3x;
                accept;
            }
        }
        term af41 {
            from {
                ether-type ipv4;
                dscp af41;
            }
            then {
                count af41;
                forwarding-class af4x;
                accept;
            }
        }
        term af4x {
            from {
                ether-type ipv4;
                dscp [ af42 af43 ];
            }
            then {
                count af4x;
                loss-priority high;
                forwarding-class af4x;
                accept;
            }
        }
        term nc {
            from {
                ether-type ipv4;
                dscp [ cs6 cs7 ];
            }
            then {
                count nc;
                forwarding-class nc;
                accept;
            }
        }
        term ncx {
            from {
                ether-type ipv4;
                dscp [ cs1 cs2 cs3 cs4 cs5 ];
            }
            then {
                count ncx;
                loss-priority high;
                forwarding-class nc;
                accept;
            }
        }
        term then_be {
            then {
                count be;
                forwarding-class be;
                accept;
            }
        }
    }
```

To recap, the previous configuration and command output displays confirm an IPv4 and Layer 2 bridged network with the connectivity shown in the test topology, and that the majority of the configuration needed for multilevel CoS are in place. Namely, definition of forwarding classes, the mapping of the same to queues, L3 and L2 BA and MF classification, and L2 and L3 BA rewrite to convey the local node's classification to downstream nodes.

Granted, the initial CoS infrastructure part is pretty easy, and again is generally part of a consistent CoS configuration baseline that's repeated in all nodes, allowing you to copy and paste the work done at node 1 with only the interface-specific parts of the CoS stanza needing to be specific to each node. Things get a bit more interesting in the next section as we move into the scheduler and scheduler map definitions.

The Scheduler Block. Getting to this point was pretty straightforward. Now comes the cerebral part. The options available for schedulers and traffic control profiles (TCPs) yield so many permutations that it's guaranteed no one size can fit all, and that is without even bringing H-CoS complexity into the mix. Junos CoS is so flexible that in many cases the same overall effects are possible using different configuration approaches. In the end, what's important is that the operation matches your network's needs, and that your CoS model is based on a constant provisioning approach to ensure that new services are turned up correctly and to ease troubleshooting and support burdens on support staff.

> Once you have a CoS infrastructure in place, you can tailor and tune its operation by adjusting and reapplying schedulers. For example, one set of schedulers can be used for core-facing interfaces, while another set is used for customer-facing links. In the latter case, several scheduler options may be available, with the set that is provisioned a function of the service level the user has signed on for.

> The modular nature of Junos CoS means that moving a user from a best effort to a business class CoS profile or increasing throughput rates from 1 Mbps to 10 Mbps requires only a few changes to the subscriber's CoS settings, in most cases using preconfigured schedulers/scheduler maps that match the service-level options. For example, several real-time schedulers can be provisioned, with the difference being transmit rate and queue depth, and possible drop profile variance, for example a gold, silver, and bronze EF scheduler. To upgrade a user's level of service, all that is needed is a modified scheduler map to reference the desired scheduler.

Such an approach is not always possible given the reality of varying CoS capabilities in the Juniper product line based on hardware type and software version. Trio platforms made a clean break from historic CoS behavior. While this may mean one more set of provisioning procedures and CoS operational quirks to have to track and learn, the trend to migrate to Trio holds the promise of an all-Trio network that offers a single, consistent CoS behavior, something we can all hope for to be sure.

As a reminder, the previously stated scheduling goals for the CoS lab are repeated:

Provide a real-time LLQ for voice and video with 25 milliseconds of delay per node and 30% of interface bandwidth. Ensure this queue cannot cause starvation for other classes.

Ensure that network control queue has priority over all non EF queues, again ensure no starvation for other classes.

Support all four Assured-Forwarding (AF) classes according to IP DiffServ.

Provide a penalty box queue for bad customers or traffic.

Share excess bandwidth among all eligible FCs.

When it comes to schedulers, the old saying about there being more than one way to skin a cat certainly holds true. There are multiple ways you could meet the stated requirements, so after much careful thought and deliberation, the plan settles on the following schedulers:

```
{master}[edit]
jnpr@R1-RE0# show class-of-service schedulers
sched_ef_30 {
    transmit-rate {
        percent 30;
        rate-limit;
    }
    buffer-size temporal 25k;
    priority strict-high;
}
sched_af4x_30 {
    transmit-rate percent 30;
    excess-rate percent 30;
    drop-profile-map loss-priority low protocol any drop-profile dp-af41;
    drop-profile-map loss-priority high protocol any drop-profile dp-af42-af43;
}
sched_af3x_15 {
    transmit-rate percent 15;
    excess-rate percent 15;
    drop-profile-map loss-priority low protocol any drop-profile dp-af31;
    drop-profile-map loss-priority high protocol any drop-profile dp-af32-af33;
}
sched_af2x_10 {
    transmit-rate percent 10;
    excess-rate percent 10;
    drop-profile-map loss-priority low protocol any drop-profile dp-af21;
    drop-profile-map loss-priority high protocol any drop-profile dp-af22-af23;
}
sched_af1x_5 {
    transmit-rate percent 5;
    excess-rate percent 5;
    drop-profile-map loss-priority low protocol any drop-profile dp-af11;
    drop-profile-map loss-priority high protocol any drop-profile dp-af12-af13;
}
sched_be_5 {
    transmit-rate percent 5;
    excess-rate percent 35;
    drop-profile-map loss-priority any protocol any drop-profile dp-be;
}
sched_nc_5 {
    transmit-rate percent 5;
    excess-rate percent 5;
    excess-priority low;
    buffer-size percent 10;
    priority high;
}
sched_null {
    priority medium-high;
    excess-priority none;
}
```

Some observations about this critical component of Junos CoS are certainly warranted here. Note there is one scheduler for each forwarding class. The null scheduler is an example of one CoS extreme, which here serves as a penalty box queue, given it has no guaranteed bandwidth, nor does it have the ability to use any excess bandwidth. The null class requires a guaranteed priority that is higher than GL to be able to use excess none, so it's set to medium, but this means little as it has no weight. This queue is where bad packets go to slowly live out their TTLs in peace. Well, that or to meet a swift and merciful end at the hands of WRED performing drops at the tail end of a mighty short queue.

The EF and NC queues, on the other hand, stand in stark contrast. EF with its SH priority must only compete with the NC queue (the only other high-priority queue), and it can never go negative because the queue is rate limited (not shaped, as buffering is bad for real-time), therefore there is no need to configure excess sharing parameters. Queue 5/EF gets up to 30% of an interface's (PIR) bandwidth and then it's policed. The buffer for this queue is limited by a temporal value, in microseconds, to control the maximum per node queuing delay to 25 milliseconds or less. Combined with the high scheduling priority and rate limiting, this creates a Low-Latency Queue (LLQ). The NC queue is at high priority (GH), which from a scheduling viewpoint is just as high as strict-high, but this scheduler is subjected to its transmit rate, and consequently has a chance for excess bandwidth, hence the inclusion of parameters to control its usage of excess bandwidth.

The NC class gets at least a guaranteed 5%, plus at least another 5% of excess bandwidth, assuming that there is some excess available. The queue is not rate limited or shaped, so it can also use additional excess that is not being used by other queues, up to IFD shaping rate. In this case, the NC scheduler has been explicitly set for excess-low priority, which is important here, because by default it would have inherited excess-high as a result of its high scheduling priority. Had this happened, the NC class could have used all excess bandwidth as it does not have a shaping rate (again, the excess rate parameter defines a *minimum* fair share, *not a maximum* usage cap), and it would have been the only queue with excess high; had this been the case, a shaper for the NC queue would have been a good idea. No shaping is used in this case because, with the settings shown, the NC queue must now contend with five others for excess bandwidth based on a WRR algorithm that's weighted based on the queue's transmit rate. Despite there being eight queues, the NC contends with only five other queues for excess bandwidth in this example because the EF scheduler cannot enter the excess region and the null class is prohibited from using any excess.

The BE scheduler is remarkable only by virtue of it having a larger weighting for excess bandwidth; the other excess eligible schedulers have an excess rate percentage set to match the queue's transmit rate. This decision was made because the BE class is given a relatively low transmit percentage (or a low guaranteed rate), and so it's expected to be sending in excess most of the time. In the worst case, when all classes are at their assigned rates, BE is only guaranteed 5% of interface speed or shaped rate. It is what it is, as they say. You have to pay to play, else it's BE for you, along with few online gaming friends and the unwelcome moniker of "lagger"!

This BE scheduler links to a single WRED profile because the concept of high versus low loss priority for *best effort* made no sense to the network architect. Again, reasonable people can disagree. The EF scheduler has no WRED profile either; its high priority and limited queue depth, combined with rate limiting, creates a LLQ, which means at most only one or two packets should be queued there, resulting in little chance for appreciable delay buffer fill, which translates to no little benefit or need for WRED. Besides, real-time applications don't respond to TCP-based implicit congestion notification anyway.

The remaining schedulers serve variations of the Assured Forwarding (AF) class. There can be up to four AF classes, and each should provide a higher level of service then the one below, making AF2 better then AF1, and AF4 the best of all. Further, according to DiffServ specifications, within each class there must be at least two loss probabilities. At a minimum, the AFx1 group must have less loss than the AFx2 or AFx3 groups. To meet these requirements, each AF class is linked to two WRED profiles; the first is used for the lower drop probability, while the latter is used for other two subclasses, giving them both the same (higher) drop probability.

Combined with the higher transmit rate for each success AF class, the different drop behaviors should offer class differentiation, which is the whole idea of CoS, assuming you get a chance to stop and smell the buffers along the way.

As a final note, all queues expected to be eligible for excess have an excess rate explicitly set, even though it matches the no-config defaults. Recall that once an excess rate is set for one queue, the defaults are off and all others get 0 excess rate unless they too are given an explicit value.

The drop profiles for the AF1x class are shown:

```
jnpr@R1-RE0# show class-of-service drop-profiles dp-af11
interpolate {
    fill-level [ 34 100 ];
    drop-probability [ 0 100 ];
}

{master}[edit]
jnpr@R1-RE0# show class-of-service drop-profiles dp-af12-af13
interpolate {
    fill-level [ 12 34 ];
    drop-probability [ 0 100 ];
}
```

Here, AF11 queues is set to begin dropping at 34% fill while AF12 and AF13 start dropping at a lower 12% fill level, going on to hit a rather harsh 100% drop probability at only 34%! The difference in profiles should have a clear impact on drop behavior under even moderate congestion.

As a final observation, the example shown follows best practice by specifying transmit rate as a percentage, thus allowing flexible application of the schedulers regardless of link speed or shaping rate. Also note that rates, as a percentage versus absolute, are used consistently within the same scheduling hierarchy; mixing transmit percentages and rates is supported but things are confusing enough already. The sum of transmit rates percentages sum to 100%; they cannot exceed 100%, but can sum to less, in which case you are guaranteeing some level of excess bandwidth is always available, even when all queues are at their configured transmit rates.

Likewise, the excess rate percentages also sum to 100%, a restriction that is not required, but again tends to make things simpler to understand and predict. Note that `excess-rate` can sum to over 100%, unlike transmit rate percentages.

The scheduler map is the last bit of the baseline configuration. It's here that you link schedulers to forwarding classes, and ultimately apply them to queues when the map is applied to an interface. The map is displayed:

```
{master}[edit]
jnpr@R1-RE0# show class-of-service scheduler-maps
```

```
sched_map_core {
    forwarding-class ef scheduler sched_ef_30;
    forwarding-class af4x scheduler sched_af4x_30;
    forwarding-class af3x scheduler sched_af3x_15;
    forwarding-class af2x scheduler sched_af2x_10;
    forwarding-class af1x scheduler sched_af1x_5;
    forwarding-class be scheduler sched_be_5;
    forwarding-class nc scheduler sched_nc_5;
    forwarding-class null scheduler sched_null;
}
```

No real surprises here. The CoS design makes use of eight queues, so there are eight schedulers, and the map in turn links each to a FC/queue. Including information like the associated FC name and scheduling rate in the scheduler names makes later modifications less prone to error and general CoS debugging that much easier.

The scheduler map is not yet attached as specifics of its attachment vary based on scheduling mode, as covered in the next section. At this time, the default 95/5% BE/NC scheduler is still in effect.

Select a Scheduling Mode

Trio MPCs can support three scheduling modes, but two of them require Q-based MPCs. Given that the MPCs used in the JMX lab at R1, R2, and R4 are equipped with Q capable MPCs, all scheduler modes are available:

```
{master}[edit]
jnpr@R1-RE0# run show chassis hardware
Hardware inventory:
Item             Version  Part number  Serial number     Description
Chassis                                JN111992BAFC      MX240
Midplane         REV 07   760-021404   TR5026            MX240 Backplane
FPM Board        REV 03   760-021392   KE2411            Front Panel Display
PEM 0            Rev 02   740-017343   QCS0748A002       DC Power Entry Module
Routing Engine 0 REV 07   740-013063   1000745244        RE-S-2000
Routing Engine 1 REV 07   740-013063   9009005669        RE-S-2000
CB 0             REV 03   710-021523   KH6172            MX SCB
CB 1             REV 10   710-021523   ABBM2781          MX SCB
FPC 2            REV 15   750-031088   YR7184            MPC Type 2 3D Q
. . .
```

The output confirms that port-based, per unit, and hierarchical scheduling are all possible, but just because a feature is supported does not in itself mean it's a good idea to deploy it. The current requirements don't specify any great levels of subscriber scaling; in fact, the R1-R2 link has the greatest number of IFLs currently provisioned, and there are only two! Hardly a case where IFL-Sets and the like seem justified. Yes, H-CoS could work, but in keeping with the principle of Ockham's razor, the simplest solution that meets all requirements is generally the best, and given the current design requirements that would be per unit scheduling mode.

When testing the v11.4R1 release, it was found that non-Q MPCs could be configured for per unit scheduling mode, and it appeared to work. However, at this time per unit and hierarchal scheduling modes are not officially supported on non-Q MPCs. Even if per unit appears to work you will have trouble getting JTAC support should anything go wrong, and as an untested feature, scale and performance values levels are currently unknown.

As mentioned previously, the per unit scheduling mode on the R1-R2 link is required to comply with the stated need to isolate Layer 2 traffic from Layer 3. In this mode, each unit on the R1-R2 link gets its own set of schedulers, and if desired each IFL can be shaped to limit its maximum bandwidth usage.

In contrast, per port CoS is all that is required on links that carry only one type of traffic; for example, the R2-R4 link has only one unit, and it's a Layer 3 IP unit, which makes the presence of native bridged impossible. Note that both per unit and H-CoS modes require that the IFL have multiple units (i.e., two VLAN tags), or you get a commit error.

Given these points, it does seem possible to meet the CoS design requirements with most of the interfaces running in port mode, which means that unless future growth in CoS scale or capabilities is planned, less expensive...but keep this in mind: if there is one thing that stays the same in IP networks, it's growth and evolution, so having hardware capabilities that may not be needed until a future time can be a sound strategy for future-proofing your network.

The multiservice interfaces at R1 and R2 are set for per unit scheduling mode, a configuration that occurs at the IFD level under the `[edit interfaces <interface-name>]` hierarchy:

```
{master}[edit]
jnpr@R1-RE0# set interfaces xe-2/0/0 per-unit-scheduler
```

The remaining interfaces are left in their default per-port mode setting.

Apply Schedulers and Shaping. With scheduling mode set, it's time to apply the schedulers to R1's egress interface, the last step to putting this massive CoS configuration into effect. The `sched_map_core` scheduler map can be applied directly to the interface IFLs using a `set class-of-service interfaces xe-2/0/0 unit 0 scheduler-map sched_map_core` statement, but you can also link to the map through a Traffic Control Profile (TCP), which offers the added benefits of allowing you to specify guaranteed, shaping, or excess rates.

This example makes use of shaping for two reasons; first, to meet the stated traffic separation, and secondly, to slow things down, so to speak, as CoS is infinitely more needed, and therefore testable, on slow speed links where it's far easier to generate congestion in order to observe the magic that is CoS.

To meet the first shaping goal, two TCPs are defined, one for each of the core interface's IFLs. Both reference the same scheduler map (different maps are supported, but not needed in this case), and both have a 5 Mbps shaping rate. The latter part is critical to ensuring the required isolation between the bridged and routed traffic, as assuming the underlying IFD supports at least 10 Mbps, both types of traffic can operate at their G-Rates simultaneously, and either can burst to its shaped rate when the other is not at full capacity:

```
{master}[edit class-of-service]
jnpr@R1-RE0# show traffic-control-profiles
tc_l2_ifl_5m {
    scheduler-map sched_map_core;
    shaping-rate 10m;
    guaranteed-rate 5m;
}
tc_l3_ifl_5m {
    scheduler-map sched_map_core;
    shaping-rate 10m;
    guaranteed-rate 5m;
}
tc_ifd_10m {
    shaping-rate 10m;
}
```

The `tc_ifd_10m` TCP is defined to shape the underlying IFD to 10 Mbps, in keeping with the plan to slow things down to make CoS easier to demonstrate and test. The TCPs are then applied to R1's core interface:

```
{master}[edit]
jnpr@R1-RE0# show class-of-service interfaces xe-2/0/0
output-traffic-control-profile tc_ifd_10m;
unit 0 {
    output-traffic-control-profile tc_l2_ifl_5m;
    classifiers {
        ieee-802.1 ieee_classify;
    }
    rewrite-rules {
        ieee-802.1 ieee_rewrite;
    }
}
unit 1 {
    output-traffic-control-profile tc_l3_ifl_5m;
    classifiers {
        dscp dscp_diffserv;
    }
    rewrite-rules {
        dscp dscp_diffserv;
    }
}
```

Clearly, with a 10 Mbps bottleneck, both traffic types cannot hope to meet their PIR/shaped rates simultaneously. In contrast, it's expected that both can operate at their CIR/G-Rates, and that no amount of user traffic should be able to starve out network control, behavior that should be easy to test and confirm. In addition, no amount of traffic on any one unit should have an impact on another unit's ability to achieve at least its G-Rate (albeit perhaps with no excess left to share).

The remaining routers, which are in per port scheduling mode as you will recall, are not shaped and left to run at their native 10 Gbps rate to ensure no congestion can occur there, a condition that if allowed would cloud up the ability to test the specific behavior of the R1-R2 link, which is the focus here. The lack of shaping, G-Rate, or excess rate specification on these routers means that a TCP cannot be used; instead the direct scheduler map linking method is used. The configuration of R2's R4 facing core interface is shown:

```
{master}[edit]
jnpr@R2-RE0# show class-of-service interfaces xe-2/1/0
scheduler-map sched_map_core;
unit 0 {
    classifiers {
        dscp dscp_diffserv;
    }
    rewrite-rules {
        dscp dscp_diffserv;
    }
}
```

Note that per port operation can be gleaned by the map's application to the IFD, rather than at the unit level.

With the completed CoS baseline now in place, again, for unidirectional traffic from the sources to the receivers, we can proceed to operational verification.

Verify Unidirectional CoS

Before starting any traffic, the application of the TCPs and BA classification/rewrite rules to R1's core interface is verified:

```
{master}[edit]
jnpr@R1-RE0# run show class-of-service interface xe-2/0/0
Physical interface: xe-2/0/0, Index: 148
Queues supported: 8, Queues in use: 8
Total non-default queues created: 24
  Output traffic control profile: tc_ifd_10m, Index: 10734
  Congestion-notification: Disabled

  Logical interface: xe-2/0/0.0, Index: 332, Dedicated Queues: yes
    Object                Name                  Type                    Index
    Traffic-control-profile tc_l2_ifl_5m        Output                  55450
    Rewrite               ieee_rewrite           ieee8021p (outer)       16962
    Classifier            ieee_classify         ieee8021p               22868

  Logical interface: xe-2/0/0.1, Index: 333, Dedicated Queues: yes
    Object                Name                  Type                    Index
    Traffic-control-profile tc_l3_ifl_5m        Output                  55442
    Rewrite               dscp_diffserv         dscp                    23080
    Classifier            dscp_diffserv         dscp                    23080
  Logical interface: xe-2/0/0.32767, Index: 334, Dedicated Queues: yes
    Object                  Name                   Type                    Index
    Traffic-control-profile __control_tc_prof      Output                  45866
```

Use the comprehensive switch to the show class-of-service interface command to detailed CoS-related information, including queue counts, drop statistics, drop profiles, and so on. The output is not shown here to save space—that's how much there is!

The output confirms the two user-configured units in addition to the automatically created unit for control traffic. Interestingly, as a result, a total of 24 queues are now allocated to the IFD, even though the control unit only has two FCs in use, thus proving allocation of queues in units of eight. The IFD- and IFL-level output TCPs are also confirmed to be in effect. Currently, input TCPs (and queuing) are not supported on Trio, but the capability is in the hardware so a future Junos release will likely offer support for ingress CoS functionality. The TCP rates are also verified:

```
{master}[edit]
jnpr@R1-RE0# run show class-of-service traffic-control-profile
Traffic control profile: tc_ifd_10m, Index: 10734
  Shaping rate: 10000000
  Scheduler map: <default>

Traffic control profile: tc_l2_ifl_5m, Index: 55450
  Shaping rate: 10000000
  Scheduler map: sched_map_core
  Guaranteed rate: 5000000

Traffic control profile: tc_l3_ifl_5m, Index: 55442
  Shaping rate: 10000000
  Scheduler map: sched_map_core
  Guaranteed rate: 5000
Confirm Queueing and Classification
```

So far there are no surprises, as the outputs match both expectations and the related configuration settings.

Confirm Queuing and Classification

Previous displays confirmed eight queues and that classifiers and rewrite rules are attached. Displaying CoS interface queues is an invaluable way to monitor and troubleshoot CoS operation. With no test traffic flowing, the interface statistics are cleared, and the egress queue information is displayed for R1's core interface (ingress stats for traffic received from the L3 source are not yet supported):

```
{master}[edit]
jnpr@R1-RE0# run show interfaces queue xe-2/0/0
Physical interface: xe-2/0/0, Enabled, Physical link is Up
  Interface index: 148, SNMP ifIndex: 4373
```

```
Forwarding classes: 16 supported, 8 in use
Egress queues: 8 supported, 8 in use
Queue: 0, Forwarding classes: be
  Queued:
    Packets              :                     0                     0 pps
    Bytes                :                     0                     0 bps
  Transmitted:
    Packets              :                     0                     0 pps
    Bytes                :                     0                     0 bps
    Tail-dropped packets :                     0                     0 pps
    RED-dropped packets  :                     0                     0 pps
     Low                 :                     0                     0 pps
     Medium-low          :                     0                     0 pps
     Medium-high         :                     0                     0 pps
     High                :                     0                     0 pps
    RED-dropped bytes    :                     0                     0 bps
     Low                 :                     0                     0 bps
     Medium-low          :                     0                     0 bps
     Medium-high         :                     0                     0 bps
     High                :                     0                     0 bps
Queue: 1, Forwarding classes: af1x
  Queued:
    Packets              :                     0                     0 pps
    Bytes                :                     0                     0 bps
  Transmitted:
    Packets              :                     0                     0 pps
    Bytes                :                     0                     0 bps
    Tail-dropped packets :                     0                     0 pps
    RED-dropped packets  :                     0                     0 pps
     Low                 :                     0                     0 pps
     Medium-low          :                     0                     0 pps
     Medium-high         :                     0                     0 pps
     High                :                     0                     0 pps
    RED-dropped bytes    :                     0                     0 bps
     Low                 :                     0                     0 bps
     Medium-low          :                     0                     0 bps
     Medium-high         :                     0                     0 bps
     High                :                     0                     0 bps
Queue: 2, Forwarding classes: af2x
  Queued:
    Packets              :                     0                     0 pps
    Bytes                :                     0                     0 bps
  Transmitted:
    Packets              :                     0                     0 pps
    Bytes                :                     0                     0 bps
    Tail-dropped packets :                     0                     0 pps
    RED-dropped packets  :                     0                     0 pps
     Low                 :                     0                     0 pps
     Medium-low          :                     0                     0 pps
     Medium-high         :                     0                     0 pps
     High                :                     0                     0 pps
    RED-dropped bytes    :                     0                     0 bps
     Low                 :                     0                     0 bps
     Medium-low          :                     0                     0 bps
     Medium-high         :                     0                     0 bps
```

```
      High                 :                    0                     0 bps
   Queue: 3, Forwarding classes: nc
     Queued:
       Packets              :                 3568                    10 pps
       Bytes                :               335795                  9160 bps
     Transmitted:
       Packets              :                 3568                    10 pps
       Bytes                :               335795                  9160 bps
       Tail-dropped packets :                    0                     0 pps
       RED-dropped packets  :                    0                     0 pps
        Low                 :                    0                     0 pps
        Medium-low          :                    0                     0 pps
        Medium-high         :                    0                     0 pps
        High                :                    0                     0 pps
       RED-dropped bytes    :                    0                     0 bps
        Low                 :                    0                     0 bps
        Medium-low          :                    0                     0 bps
        Medium-high         :                    0                     0 bps
        High                :                    0                     0 bps
   Queue: 4, Forwarding classes: af4x
     Queued:
       Packets              :                    0                     0 pps
       Bytes                :                    0                     0 bps
     Transmitted:
       Packets              :                    0                     0 pps
       Bytes                :                    0                     0 bps
       Tail-dropped packets :                    0                     0 pps
       RED-dropped packets  :                    0                     0 pps
        Low                 :                    0                     0 pps
        Medium-low          :                    0                     0 pps
        Medium-high         :                    0                     0 pps
        High                :                    0                     0 pps
       RED-dropped bytes    :                    0                     0 bps
        Low                 :                    0                     0 bps
        Medium-low          :                    0                     0 bps
        Medium-high         :                    0                     0 bps
        High                :                    0                     0 bps
   Queue: 5, Forwarding classes: ef
     Queued:
       Packets              :                    0                     0 pps
       Bytes                :                    0                     0 bps
     Transmitted:
       Packets              :                    0                     0 pps
       Bytes                :                    0                     0 bps
       Tail-dropped packets :                    0                     0 pps
       RED-dropped packets  :                    0                     0 pps
        Low                 :                    0                     0 pps
        Medium-low          :                    0                     0 pps
        Medium-high         :                    0                     0 pps
        High                :                    0                     0 pps
       RED-dropped bytes    :                    0                     0 bps
        Low                 :                    0                     0 bps
        Medium-low          :                    0                     0 bps
        Medium-high         :                    0                     0 bps
        High                :                    0                     0 bps
```

```
Queue: 6, Forwarding classes: af3x
  Queued:
    Packets              :                     0                     0 pps
    Bytes                :                     0                     0 bps
  Transmitted:
    Packets              :                     0                     0 pps
    Bytes                :                     0                     0 bps
    Tail-dropped packets :                     0                     0 pps
    RED-dropped packets  :                     0                     0 pps
     Low                 :                     0                     0 pps
     Medium-low          :                     0                     0 pps
     Medium-high         :                     0                     0 pps
     High                :                     0                     0 pps
    RED-dropped bytes    :                     0                     0 bps
     Low                 :                     0                     0 bps
     Medium-low          :                     0                     0 bps
     Medium-high         :                     0                     0 bps
     High                :                     0                     0 bps
Queue: 7, Forwarding classes: null
  Queued:
    Packets              :                     0                     0 pps
    Bytes                :                     0                     0 bps
  Transmitted:
    Packets              :                     0                     0 pps
    Bytes                :                     0                     0 bps
    Tail-dropped packets :                     0                     0 pps
    RED-dropped packets  :                     0                     0 pps
     Low                 :                     0                     0 pps
     Medium-low          :                     0                     0 pps
     Medium-high         :                     0                     0 pps
     High                :                     0                     0 pps
    RED-dropped bytes    :                     0                     0 bps
     Low                 :                     0                     0 bps
     Medium-low          :                     0                     0 bps
     Medium-high         :                     0                     0 bps
     High                :                     0                     0 bps
```

As expected, only network control is currently flowing; it's quite rare to have a network so entirely under one's control that CoS can be tested in this granular a manner. Ah, the beauty of a test lab. Enjoy it while it lasts. Given the length, subsequent output queue displays will focus on one class or another based on what is being tested at the time.

Use Ping to Test MF Classification. The MF classifier is confirmed by generating some EF marked test traffic from S1 to the L2 receiver:

```
{master:0}[edit]
jnpr@S1-RE0# run ping 192.0.2.7 tos 184 count 5 rapid
PING 192.0.2.7 (192.0.2.7): 56 data bytes
!!!!!
--- 192.0.2.7 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max/stddev = 1.014/1.671/4.066/1.200 ms
```

```
{master:0}[edit]
jnpr@S1-RE0#
```

And the egress queue counts at R1 confirm all went to plan, at least at the first hop:

```
{master}[edit]
jnpr@R1-RE0# run show interfaces queue xe-2/0/0 forwarding-class ef
Physical interface: xe-2/0/0, Enabled, Physical link is Up
  Interface index: 148, SNMP ifIndex: 4373
Forwarding classes: 16 supported, 8 in use
Egress queues: 8 supported, 8 in use
Queue: 5, Forwarding classes: ef
  Queued:
    Packets              :                     5                     0 pps
    Bytes                :                   630                     0 bps
  Transmitted:
    Packets              :                     5                      0 pps
    Bytes                :                   630                     0 bps
    Tail-dropped packets :                     0                     0 pps
    RED-dropped packets  :                     0                     0 pps
     Low                 :                     0                     0 pps
     Medium-low          :                     0                     0 pps
     Medium-high         :                     0                     0 pps
     High                :                     0                     0 pps
    RED-dropped bytes    :                     0                     0 bps
     Low                 :                     0                     0 bps
     Medium-low          :                     0                     0 bps
     Medium-high         :                     0                     0 bps
     High                :                     0                     0 bps
```

Consistent BA-based classification, and therefore marker write, is verified at the downstream node by repeating the ping at S1 after clearing statistics at R2:

```
{master}[edit]
jnpr@R2-RE0# run show interfaces queue xe-2/2/0 forwarding-class ef
Physical interface: xe-2/2/0, Enabled, Physical link is Up
  Interface index: 183, SNMP ifIndex: 665
Forwarding classes: 16 supported, 8 in use
Egress queues: 8 supported, 8 in use
Queue: 5, Forwarding classes: ef
  Queued:
    Packets              :                     5                     0 pps
    Bytes                :                   630                     0 bps
  Transmitted:
    Packets              :                     5                     0 pps
    Bytes                :                   630                     0 bps
    Tail-dropped packets :                     0                     0 pps
    RED-dropped packets  :                     0                     0 pps
     Low                 :                     0                     0 pps
     Medium-low          :                     0                     0 pps
     Medium-high         :                     0                     0 pps
     High                :                     0                     0 pps
    RED-dropped bytes    :                     0                     0 bps
     Low                 :                     0                     0 bps
     Medium-low          :                     0                     0 bps
     Medium-high         :                     0                     0 bps
     High                :                     0                     0 bps
```

Again, the count matches the generated test traffic nicely. As a final check of end-to-end classification and rewrite, an AF42 marked ping is generated by R1 to the L3 receiver:

```
{master}[edit]
jnpr@R1-RE0# run ping 192.168.4.1 rapid count 69 tos 144
PING 192.168.4.1 (192.168.4.1): 56 data bytes
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
--- 192.168.4.1 ping statistics ---
69 packets transmitted, 69 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.705/1.317/35.678/4.203 ms
```

And (having first cleared the queue counters as previously mentioned), the egress count, this time at R4 is displayed, and again found to be as expected, confirming L3 classification is working end-to-end:

```
[edit]
jnpr@R4# run show interfaces queue xe-2/2/0 forwarding-class af4x
Physical interface: xe-2/2/0, Enabled, Physical link is Up
  Interface index: 152, SNMP ifIndex: 544
Forwarding classes: 16 supported, 8 in use
Egress queues: 8 supported, 8 in use
Queue: 4, Forwarding classes: af4x
  Queued:
    Packets              :                    69                     0 pps
    Bytes                :                  8418                     0 bps
  Transmitted:
    Packets              :                    69                     0 pps
    Bytes                :                  8418                     0 bps
    Tail-dropped packets :                     0                     0 pps
    RED-dropped packets  :                     0                     0 pps
     Low                 :                     0                     0 pps
     Medium-low          :                     0                     0 pps
     Medium-high         :                     0                     0 pps
     High                :                     0                     0 pps
    RED-dropped bytes    :                     0                     0 bps
     Low                 :                     0                     0 bps
     Medium-low          :                     0                     0 bps
     Medium-high         :                     0                     0 bps
     High                :                     0                     0 bps
```

Useful CLI ToS Mappings

The following is a mapping of IP precedence to binary, along with the resulting decimal equivalent. This can be useful when testing CoS using utilities such as ping or traceroute. In Junos when you include the tos argument to a ping or traceroute, you must specify the desired ToS coding using *decimal*, not binary or hexadecimal.

NC:	DSCP		Decimal
Precedence 7 →	111000xx -->	128+64+32+0+0+0+x+x	→ 224

Precedence 6 →	110000xx →	128+64+0+0+0+0+x+x	→ 192
Precedence 5 →	101000xx →	128+0+32+0+0+0+x+x	→ 160
Precedence 4 →	100000xx →	128+0+0+0+0+0+x+x	→ 128
Precedence 3 →	011000xx →	0+64+32+0+0+0+x+x	→ 96
Precedence 2 →	010000xx →	0+64+0+0+0+0+x+x	→ 64
Precedence 1 →	010000xx →	0+0+32+0+0+0+x+x	→ 32
EF:			
DSCP EF →	101110xx →	128+0+32+16+8+0+x+x	→ 184
AF:			
DSCP AF12 →	001100xx →	0+0+32+16+0+0+x+x	→ 48
DSCP AF22 →	010100xx →	0+64+0+16+0+0+x+x	→ 80
DSCP AF32 →	011100xx →	0+64+32+16+0+0+x+x	→ 112
DSCP AF42 →	100100xx →	128+0+0+16+0+0+x+x	→ 144

Also, be sure to use the `show class-of-service code-point-aliases` command to see mappings of BAs to FCs.

Previous results confirm that both L2 and L3 traffic is properly classified end-to-end, at least when constrained to their native domains. Before moving on, a final test of classification is performed to verify Layer 2 to Layer 3 classification/rewrite. As before, interface counters are cleared (now back at R1), and test traffic is again generated from S1, this time with an EF ToS and now destined for the L3 receiver, thus forcing traversal of the IRB at R1:

```
{master:0}[edit]
jnpr@S1-RE0# run ping 192.168.4.1 rapid count 100 tos 184
PING 192.168.4.1 (192.168.4.1): 56 data bytes
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
--- 192.168.4.1 ping statistics ---
100 packets transmitted, 100 packets received, 0% packet loss
round-trip min/avg/max/stddev = 1.008/1.522/3.950/0.954 ms
```

Meanwhile, at R1, the news is not very cheerful:

```
{master}[edit]
jnpr@R1-RE0# run show interfaces queue xe-2/0/0
Physical interface: xe-2/0/0, Enabled, Physical link is Up
  Interface index: 148, SNMP ifIndex: 4373
Forwarding classes: 16 supported, 8 in use
Egress queues: 8 supported, 8 in use
Queue: 0, Forwarding classes: be
  Queued:
    Packets              :                   100                     0 pps
    Bytes                :                 12600                     0 bps
  Transmitted:
    Packets              :                   100                     0 pps
    Bytes                 :                 12600                     0 bps
    Tail-dropped packets :                     0                     0 pps
    RED-dropped packets  :                     0                     0 pps
     Low                 :                     0                     0 pps
. . .
```

Clearly, classification is broken for the interdomain traffic as it was all queued as BE. The MF classification filter is cleared, and the test is repeated:

```
{master}[edit]
jnpr@R1-RE0# run clear firewall all

<pings at S1 ommited for brevity>

{master}[edit]
jnpr@R1-RE0# run show firewall

Filter: __default_bpdu_filter__

Filter: l2_mf_classify
Counters:
Name                                                Bytes              Packets
af11                                                    0                    0
af1x                                                    0                    0
af21                                                    0                    0
af2x                                                    0                    0
af31                                                    0                    0
af3x                                                    0                    0
af41                                                    0                    0
af4x                                                    0                    0
be                                                      0                    0
ef                                                  10200                  100
```

The EF term count matching test traffic confirms that the filter did its job. This is interesting, as the same filter and test traffic was used for intra-L2 traffic and found to work. Likewise, L3 classification from R1 to R4 was also confirmed. Clearly, there is some piece in the middle not yet tested. That piece is the IRB interface itself, as it's the glue used to interconnect L2 and L3 domains. Having the wrong classifier there would only affect traffic flowing between L2 and L3 domains, just as observed. The IRB's CoS settings are displayed and found to be wanting:

```
{master}[edit]
jnpr@R1-RE0# run show class-of-service interface irb
Physical interface: irb, Index: 142
Queues supported: 8, Queues in use: 8
  Scheduler map: <default>, Index: 2
  Congestion-notification: Disabled

  Logical interface: irb.100, Index: 321
    Object                  Name                   Type                    Index
    Classifier              ipprec-compatibility   ip                         13

  Logical interface: irb.200, Index: 325
    Object                  Name                   Type                    Index
    Classifier              ipprec-compatibility   ip                         13
```

That looks like a default CoS profile, which supports only BE and NC, and thus goes far to explain why DiffServ-based classification went so wrong once the interface was crossed. The test results confirm that ingress BA/MF classification is lost when traffic transits an IRB, so be sure to apply classifiers, either BA or MF, to the IRB interface when supporting multilevel CoS. Generally, you do not need BA rewrite rules on the IRB interface as the rewrite occurs at egress from the Layer 2 or Layer 3 interface, depending on directionality. In this example, a rewrite rule is attached to keep the IRB interface consistent with others:

```
{master}[edit]
jnpr@R1-RE0# show | compare
[edit class-of-service interfaces]
+    irb {
+        unit 100 {
+            classifiers {
+                dscp dscp_diffserv;
+            }
+            rewrite-rules {
+                dscp dscp_diffserv;
+            }
+        }
+    }
```

The change is committed, counters again cleared at R1, and the EF ping is again performed at S1:

```
{master}[edit]
jnpr@R1-RE0# run clear interfaces statistics all

<EF ping repeated at S1>

{master}[edit]
jnpr@R1-RE0# run show interfaces queue xe-2/0/0 forwarding-class ef
Physical interface: xe-2/0/0, Enabled, Physical link is Up
  Interface index: 148, SNMP ifIndex: 4373
Forwarding classes: 16 supported, 8 in use
Egress queues: 8 supported, 8 in use
Queue: 5, Forwarding classes: ef
  Queued:
    Packets              :                   100                     0 pps
    Bytes                :                 12600                     0 bps
  Transmitted:
    Packets              :                   100                     0 pps
    Bytes                :                 12600                     0 bps
    Tail-dropped packets :                     0                     0 pps
    RED-dropped packets  :                     0                     0 pps
     Low                 :                     0                     0 pps
     Medium-low          :                     0                     0 pps
     Medium-high         :                     0                     0 pps
     High                :                     0                     0 pps
    RED-dropped bytes    :                     0                     0 bps
     Low                 :                     0                     0 bps
     Medium-low          :                     0                     0 bps
     Medium-high         :                     0                     0 bps
     High                :                     0                     0 bps
```

At this time, Junos does not support or need scheduling on IRB interfaces. Only MF/BA classification and rewrite rules are supported.

Great, just the results that were hoped for! DiffServ classification and rewrite is confirmed working for Layer 2 traffic, Layer 3 traffic, and for *brouted* traffic (had to throw that word in there), which in this case is traffic that is switched from Layer 2 into Layer 3 for routing.

Confirm Scheduling Details

Next is the confirmation of scheduler configuration, and how they are mapped to queues/FCs. First, the CLI commands:

```
{master}[edit]
jnpr@R1-RE0# run show class-of-service scheduler-map sched_map_core
Scheduler map: sched_map_core, Index: 20205

  Scheduler: sched_be_5, Forwarding class: be, Index: 4674
    Transmit rate: 5 percent, Rate Limit: none, Buffer size: remainder,
      Buffer Limit: none, Priority: low
    Excess Priority: unspecified, Excess rate: 35 percent,
    Drop profiles:
      Loss priority   Protocol    Index    Name
      Low             any         48733    dp-be
      Medium low      any         48733    dp-be
      Medium high     any         48733    dp-be
      High            any         48733    dp-be

  Scheduler: sched_af1x_5, Forwarding class: af1x, Index: 12698
    Transmit rate: 5 percent, Rate Limit: none, Buffer size: remainder,
      Buffer Limit: none, Priority: low
    Excess Priority: unspecified, Excess rate: 5 percent,
    Drop profiles:
      Loss priority   Protocol    Index    Name
      Low             any         64745    dp-af11
      Medium low      any             1    <default-drop-profile>
      Medium high     any             1    <default-drop-profile>
      High            any         51467    dp-af12-af13

  Scheduler: sched_af2x_10, Forwarding class: af2x, Index: 13254
    Transmit rate: 10 percent, Rate Limit: none, Buffer size: remainder,
      Buffer Limit: none, Priority: low
    Excess Priority: unspecified, Excess rate: 10 percent,
    Drop profiles:
      Loss priority   Protocol    Index    Name
      Low             any         64649    dp-af21
      Medium low      any             1    <default-drop-profile>
      Medium high     any            1    <default-drop-profile>
      High            any         2411    dp-af22-af23

  Scheduler: sched_nc_5, Forwarding class: nc, Index: 2628
    Transmit rate: 5 percent, Rate Limit: none, Buffer size: 10 percent,
      Buffer Limit: none, Priority: high
    Excess Priority: low, Excess rate: 5 percent,
    Drop profiles:
      Loss priority   Protocol    Index    Name
      Low             any            1    <default-drop-profile>
      Medium low      any            1    <default-drop-profile>
      Medium high     any            1    <default-drop-profile>
      High            any            1    <default-drop-profile>

  Scheduler: sched_af4x_30, Forwarding class: af4x, Index: 13286
    Transmit rate: 30 percent, Rate Limit: none, Buffer size: remainder,
      Buffer Limit: none, Priority: low
    Excess Priority: unspecified, Excess rate: 30 percent,
    Drop profiles:
      Loss priority   Protocol    Index    Name
      Low             any         64585    dp-af41
      Medium low      any            1    <default-drop-profile>
      Medium high     any            1    <default-drop-profile>
      High            any         35242    dp-af42-af43

  Scheduler: sched_ef_30, Forwarding class: ef, Index: 51395
    Transmit rate: 30 percent, Rate Limit: rate-limit, Buffer size: 25000 us,
      Buffer Limit: exact, Priority: strict-high
    Excess Priority: unspecified
    Drop profiles:
      Loss priority   Protocol    Index    Name
      Low             any            1    <default-drop-profile>
      Medium low      any            1    <default-drop-profile>
      Medium high     any            1    <default-drop-profile>
      High            any            1    <default-drop-profile>

  Scheduler: sched_af3x_15, Forwarding class: af3x, Index: 13267
    Transmit rate: 15 percent, Rate Limit: none, Buffer size: remainder,
      Buffer Limit: none, Priority: low
    Excess Priority: unspecified, Excess rate: 15 percent,
    Drop profiles:
      Loss priority   Protocol    Index    Name
      Low             any         64681    dp-af31
      Medium low      any            1    <default-drop-profile>
      Medium high     any            1    <default-drop-profile>
      High            any         18763    dp-af32-af33

  Scheduler: sched_null, Forwarding class: null, Index: 21629
    Transmit rate: unspecified, Rate Limit: none, Buffer size: remainder,
      Buffer Limit: none, Priority: medium-high
    Excess Priority: none
    Drop profiles:
      Loss priority   Protocol    Index    Name
      Low             any            1    <default-drop-profile>
      Medium low      any            1    <default-drop-profile>
```

```
      Medium high     any             1    <default-drop-profile>
      High            any             1    <default-drop-profile>
```

With the CLI outputs displaying expected settings and values, attention shifts to the scheduler settings in the PFE itself.

> In some cases, CLI configuration values may be adjusted by either rounding up or down in order to fit into available hardware capabilities. In most cases, these adjustments are minimal and do not yield any appreciable differences in observed CoS behavior. However, sometimes the CLI configuration may be rejected; perhaps due to an unsupported configuration or lack of required hardware. Sometimes, the result of such an error is default CoS values being programmed into the PFE.
>
> In most cases, operational mode CLI commands will make this condition known, but when all else fails, and you have no idea why your configuration is not behaving the way you expected, confirming the values actually placed into the PFE is a good place to begin troubleshooting. It also wise to check for any conflicts of errors reported in the system log when any changes are made to the CoS configuration.

The scheduling hierarchy for the per unit scheduler that's now in effect on R1's xe-2/0/0 interface is displayed:

```
NPC2(R1-RE0 vty)# sho cos scheduler-hierarchy

class-of-service EGRESS scheduler hierarchy - rates in kbps
---------------------------------------------------------------------------------
                                    shaping guarntd delaybf  excess
interface name               index    rate    rate    rate    rate      other
---------------------------- -----  ------- ------- ------- ------- -------------
xe-2/0/0                       148    10000       0       0       0
  xe-2/0/0.0                   332    10000    5000       0       0
    q 0 - pri 0/0            20205        0      5%       0     35%
    q 1 - pri 0/0            20205        0      5%       0      5%
    q 2 - pri 0/0            20205        0     10%       0     10%
    q 3 - pri 3/1            20205        0      5%     10%      5%
    q 4 - pri 0/0            20205        0     30%       0     30%
    q 5 - pri 4/0            20205        0     30%   25000      0%     exact
    q 6 - pri 0/0            20205        0     15%       0     15%
    q 7 - pri 2/5            20205        0       0       0      0%
  xe-2/0/0.1                   333    10000       0       0       0
    q 0 - pri 0/0            20205        0      5%       0     35%
    q 1 - pri 0/0            20205        0      5%       0      5%
    q 2 - pri 0/0            20205        0     10%       0     10%
    q 3 - pri 3/1            20205        0      5%     10%      5%
    q 4 - pri 0/0            20205        0     30%       0     30%
    q 5 - pri 4/0            20205        0     30%   25000      0%     exact
    q 6 - pri 0/0            20205        0     15%       0     15%
    q 7 - pri 2/5            20205        0       0       0      0%
  xe-2/0/0.32767               334        0    2000    2000       0
    q 0 - pri 0/1                2        0     95%     95%      0%
```

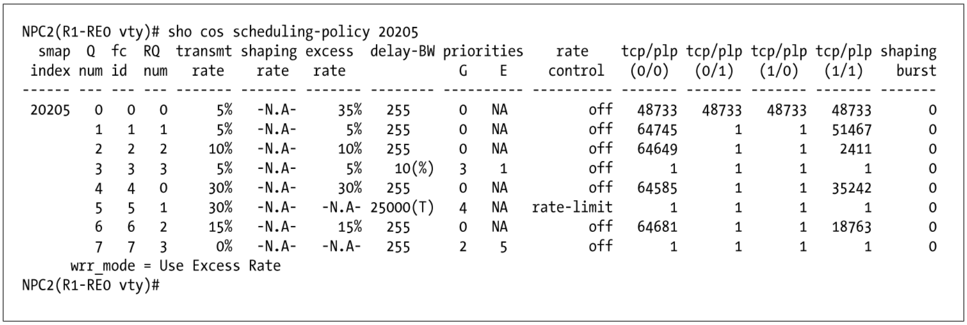
```
       q 3 - pri 0/1                2        0       5%        5%       0%
    xe-2/0/1                      149        0        0         0        0
    xe-2/1/0                      150        0        0         0        0
    xe-2/1/1                      151        0        0         0        0
    xe-2/2/0                      152        0        0         0        0
    xe-2/2/1                      153        0        0         0        0
    xe-2/3/0                      154        0        0         0        0
    xe-2/3/1                      155        0        0         0        0
```

The scheduler block initially appears as expected, apparently matching the configuration well. Two units are present, each with eight queues, plus the automatically created control 32767. The IFD shaping rate is confirmed at 10 Mbps, as are both IFL shaping rates, also at 10 Mbps. The CIR/guaranteed rate for IFL 0 correctly indicates 5 Mbps; oddly though, IFL 1 does not show a G-Rate, given the identical configuration that is not expected. Also of note is that the automatically generated control channel, which has managed to reserve 2 Mbps of G-Rate for itself. You may want to keep this tidbit in mind; there's more on this point later.

Both units have the same scheduler map, and so both sets of user queues are identical. The values shown reflect their configured parameters.

Note that queue 5, used for EF, is marked with `exact`, indicating it is rate limited (or shaped) to the specified speed/rate, and is not eligible for excess. Its configured temporal buffer is shown, as is the nondefault 10% buffer allocation made to the NC queue; all other queues use the default delay buffer based on transmit rate. The queues also display the configured share of excess rate as a percentage; here, BE gets 35% weighting for excess bandwidth, while the EF and Null queues are restricted from any excess. Note how queue 7, the Null FC, accurately reflects its lack of transmit rate and inability to use excess bandwidth.

The index number reflects the scheduling policy for the queue, and the use of the same scheduler map on both IFLs gives all queues the same value. The policy wrapped in book format was messy, so it's shown in Figure 5-29.



```
NPC2(R1-RE0 vty)# sho cos scheduling-policy 20205
  smap  Q  fc  RQ  transmt shaping excess  delay-BW priorities    rate    tcp/plp tcp/plp tcp/plp tcp/plp shaping
 index num id  num   rate    rate   rate             G    E     control   (0/0)   (0/1)   (1/0)   (1/1)    burst
------ --- --- --- ------- ------- ------- -------- ---------- ---------- ------- ------- ------- ------- -------
 20205   0   0   0      5%  -N.A-      35%   255      0   NA          off   48733   48733   48733   48733       0
         1   1   1      5%  -N.A-       5%   255      0   NA          off   64745       1       1   51467       0
         2   2   2     10%  -N.A-      10%   255      0   NA          off   64649       1       1    2411       0
         3   3   3      5%  -N.A-       5%    10(%)   3    1          off       1       1       1       1       0
         4   4   0     30%  -N.A-      30%   255      0   NA          off   64585       1       1   35242       0
         5   5   1     30%  -N.A-    -N.A- 25000(T)   4   NA   rate-limit       1       1       1       1       0
         6   6   2     15%  -N.A-      15%   255      0   NA          off   64681       1       1   18763       0
         7   7   3      0%  -N.A-    -N.A-   255      2    5          off       1       1       1       1       0
      wrr_mode = Use Excess Rate
NPC2(R1-RE0 vty)#
```

Figure 5-29. Scheduler Policy.

The output matches other displays and the configuration. Note that the priority values shown are for internal use and *not* the ones documented in this chapter, where GH through EL range from 0 to 4. The NA in the E column here indicates that no explicit excess priority is specified causing the default inheritance; the value 5 for queue 7 indicates a queue that is blocked from excess usage. Also note the various WRED profile indexes are displayed, should you care to view them.

You can view drop profiles in the CLI with the show class-of-service drop-profiles command. While we are here, the default WRED profile, with index 1, is displayed in the MPC itself:

```
NPC2(R1-RE0 vty)# sho cos red-drop-profile 1
Profile Id:     1
                 fill-level: 100% drop-probability: 100%
Wred curve configuration : 1

curve point   fill-level   drop prob
-----------   ----------   ---------
     0            0 %           0 %
     1            1 %           0 %
. . .
    62           96 %           0 %
    63           98 %         100 %
```

The output is truncated to save space, but you get the idea; with this profile, no WRED drops occur until the buffer is 100% full, effectively disabling WRED. The details of how scheduling parameter are actually programmed into platform-specific hardware is available via the CoS Hardware Abstraction Layer (HAL). This information is very useful in confirming queue-level priority and bandwidth settings. To begin, the Layer 2 IFL scheduling parameters are displayed; note the IFL index number 332 is obtained from the previous display:

```
NPC2(R1-RE0 vty)# show cos halp ifl 332
IFL type: Basic

--------------------------------------------------------------------------------
IFL name: (xe-2/0/0.0, xe-2/0/0)   (Index 332, IFD Index 148)
    QX chip id: 0
    QX chip dummy L2 index: -1
    QX chip L3 index: 3
    QX chip base Q index: 24
Queue    State        Max       Guaranteed   Burst  Weight Priorities Drop-Rules
Index                 rate         rate      size              G    E   Wred  Tail
------ ----------- ----------- ------------ ------- ------ ---------- ----------
    24  Configured    10000000       250000  131072    350   GL   EL    5     6
    25  Configured    10000000       250000  131072     50   GL   EL    6     6
    26  Configured    10000000       500000  131072    100   GL   EL    7     6
    27  Configured    10000000       250000  131072     50   GH   EL    4   195
    28  Configured    10000000      1500000  131072    300   GL   EL    8     6
    29  Configured    10000000     Disabled  131072      1   GH   EH    4   196
    30  Configured    10000000       750000  131072    150   GL   EL    9     6
    31  Configured    10000000            0  131072      1   GM   EN    4   133

Rate limit info:
    Q 5: Bandwidth = 3000000, Burst size = 300000. Policer NH: 0x30a8da3200141000
```

The output confirms the EF and NC queues are both at high scheduling priority. The Null FC is at medium while all others are at the default low. Note that the queues are shown as being shaped to the IFD rate of 10 Mbps under the max rate column. This is expected, given that no queues have a shaping rate statement of their own. By default, a queue's PIR is equal to the IFL shaping rate. If the IFL is not shaped, the IFD shaping rate or IFD rate is used.

The EF queue is rate limited, as previously noted. This is why it has a disabled G-Rate, and the details on its rate limit are shown at the bottom of this display.

You might be surprised to see the EF rate limit is 3 Mbps, which is *30% of the IFL's shaped rate* as opposed to 30% of the guaranteed rate, especially given how the other queues show a bandwidth value that is based on their transmit percentage factors against the 5 Mbps G-Rate. This is not a function of the EF class, or strict-high scheduling, but is in fact related to the use of rate limits or shaping through the use of `rate-limit` or `exact`.

> This point often catches users off guard, so it bears stressing that when you set a queue transmit rate as a percentage without including `rate-limit` or `exact`, the bandwidth for that queue is a function of the CIR, or G-Rate; if no CIR is specified, the IFL shaping rate is used. When either is specified, then a transmit rate as a percentage is based on the IFL's shaping rate, even if a G-Rate is configured. Because both are queue-level settings, it's possible to end up with a mix of transmit rates, some based on shaping rate and others on G-Rate. To work around this quirk, you can assign an absolute bandwidth or simply reduce the rate percentage by one-half.

The weight column reflects the queue's WRR share of excess bandwidth; queue 0, BE, has the largest weighting given its higher excess rate setting. In contrast, both queue 5 and queue 7 have minimal excess weight, reflecting their inability to enter and use bandwidth in the excess region. Queue 7 reflects its medium priority setting, for what that is worth, given the other settings grant it nothing, which means it has little cause to brag about its elevated priority in life! A final note regarding the display is that all queues show the same excess low priority; for most this is a function of default inheritance based on low scheduler priority. For NC, this is an explicit setting and was intended to keep it from using all available excess bandwidth, an event that can easily have happened if it was the only queue at EH with no rate limiting/shaping. Queue 5, EF, shows the default EH, but is rate limited and so again cannot enter excess. Queue 7 shows none for its priority, as expected.

Check for Any Log Errors

So far, all confirmation steps have returned the expected results, more or less. Recall there was a discrepancy noted with regard to IFL 1 not showing a G-Rate in a previous display. Perhaps that is normal, perhaps not. As always, scanning logs for error messages or reports of conflicts is a good idea, more so when an operation is found to deviate from your expectations based on a device's configuration.

A powerful feature of Junos is the ability to configure things that are not backed up with supported hardware, for example a MIC that is not yet installed, without any obvious warnings or commit errors. Again, this is a feature, in that one day the requisite hardware can be hot-inserted and boom, things just work. However, one man's feature can sometimes be seen as another's bug; it's one thing to configure a feature you know is intended for future use because that hardware is currently lacking, and it's quite another to simply not be aware of some hardware requirement or configuration restriction, only to find yourself confused when your configurations changes appear to be ignored.

Keeping to this advice, the cosd and messages logs are scanned for any errors reported around the time the per unit scheduler configuration was committed. Unfortunately, in this case the search yields fruit, only it seems to be of the rotten kind; the following is found in both the log files:

```
{master}[edit]
jnpr@R1-RE0# run show log messages | match cos
. . .
NPC2(R1-RE0 vty)# [May 21 19:44:54.439 LOG: Err] COS(cos_add_guaranteed_rate_on_ifl:
  3550): ifd(xe-2/0/0) guaranteed_bw_remain (3000000) is less than ifl-333's configured
  guaranteed rate (5000000)
. . .
```

These log messages can seem cryptic; so, like a code breaker, tweeze out each detail you can, until sense is made or you cry foul and concede to open a JTAC case. The message clearly relates to CoS on the xe-2/0/0 interface, which makes it of concern, to say the least. It goes on to complain about insufficient guaranteed rate, claiming that 3 Mbps is available (where are those commas when you need them?), and that 5 Mbps is needed. Previous displays confirm that IFL index 333 is unit 1, the Layer 3 unit, on the xe-2/0/0 interface, and it has a TCP applied with a 5 Mbps G-Rate and a 10 Mbps PIR. It's odd that no error is noted for IFL 0, as it has the same traffic control profile parameters. But where is the 3 Mbps/need 5 Mbps issue coming from? The IFD is shaped to 10 M, which should make both 5 Mbps CIRs possible.

Then the answer dawns: a previous VTY-level show scheduling-hierarchy command indicated that *three IFLs* are provisioned on this interface. Two were user configured, but the third was automatically created to pass LACP traffic, and darned if it did not get a 2 Mbps G-Rate for its scheduler. Math is a beautiful thing, at least when it works out; the 10 Mbps IFD shaping rate was first deducted 2 Mbps for the control unit's CIR, then another 5 Mbps for unit 0, the bridged unit. That left only 3 Mbps remaining, and unit 1 wants a 5 m CIR!

And viola, the issue is crystal clear. Keeping in mind that you can't overbook G-Rates on Trio when in per unit scheduler mode, the result of this error is a G-Rate imbalance between the two IFLs, with IFL 0 getting the configured G-Rate while IFL 1 gets no guaranteed rate. As such, this is not a simple matter of potential CIR overbooking, but an actual service-impacting condition!

Testing shows that in fact, significantly higher drops rates are observed on the Layer 3 IFL when both are driven simultaneously at a 10.6 Mbps rate to produce heavy congestion, thereby proving it has diminished performance when compared to IFL 0. Both IFL streams combine to produce an egress load of 21.2 Mbps at the interface, which recall is shaped to 10 Mbps. But it gets worse. After several seconds of traffic flow, it appeared the network control queue on IFL 1 was experiencing delays (but not starvation), which resulted in BFD flaps, that in turn brought down the IS-IS adjacency between R1 and R2, which caused a loss of routing and more traffic loss:

```
May 22 12:54:11  R1-RE0 rpd[1458]: RPD_ISIS_ADJDOWN: IS-IS lost L2 adjacency to R2-RE0
  on xe-2/0/0.1, reason: 3-Way Handshake Failed
May 22 12:54:11  R1-RE0 bfdd[1469]: BFDD_TRAP_SHOP_STATE_DOWN: local discriminator: 18,
  new state: down, interface: xe-2/0/0.1, peer addr: 10.8.0.1
May 22 12:54:20  R1-RE0 bfdd[1469]: BFDD_TRAP_SHOP_STATE_UP: local discriminator: 18,
  new state: up, interface: xe-2/0/0.1, peer addr: 10.8.0.1
May 22 12:54:25  R1-RE0 rpd[1458]: RPD_ISIS_ADJUP: IS-IS new L2 adjacency to R2-RE0
  on xe-2/0/0.1
May 22 12:54:53  R1-RE0 rpd[1458]: RPD_ISIS_ADJDOWN: IS-IS lost L2 adjacency to R2-RE0
  on xe-2/0/0.1, reason: 3-Way Handshake Failed
May 22 12:54:53  R1-RE0 bfdd[1469]: BFDD_TRAP_SHOP_STATE_DOWN: local discriminator: 19,
  new state: down, interface: xe-2/0/0.1, peer addr: 10.8.0.1
May 22 12:55:06  R1-RE0 rpd[1458]: RPD_ISIS_ADJUP: IS-IS new L2 adjacency to R2-RE0 on
  xe-2/0/0.1
May 22 12:55:24  R1-RE0 rpd[1458]: RPD_ISIS_ADJDOWN: IS-IS lost L2 adjacency to R2-RE0
  on xe-2/0/0.1, reason: 3-Way Handshake Failed
May 22 12:55:33  R1-RE0 rpd[1458]: RPD_ISIS_ADJUP: IS-IS new L2 adjacency to R2-RE0
  on xe-2/0/0.1
```

This is very serious indeed and helps demonstrate how a CoS error can sometimes leave you with unpredictable CoS behavior, which can be a real problem if your network relies on CoS to keep the control plane safe and secure, even under extreme congestion.

It is noted that in this case the BFD session is running at a 1 millisecond interval with a multiplier of 3. And, it bears reiterating that BFD is a protocol designed to provide *rapid detection* of forwarding plane faults, and in this case it's running with a somewhat aggressive timer.

At any extent, results show that the lack of a G-rate caused IFL 1 to lose what should have been a fair contention process for an equal share of the available 10 Mbps on the IFD, which resulted in at least 3 milliseconds too much delay for some BFD update. Put in that context the CoS issue may not seem so severe; I mean, 3 milliseconds is too much delay, really? But in this setup, it was enough to start the house of cards falling.

Several possible solutions present themselves. You could switch to H-CoS, which allows overbooking:

```
{master}[edit]
jnpr@R1-RE0# show | compare rollback 3
[edit interfaces xe-2/0/0]
```

```
-   per-unit-scheduler;
-   vlan-tagging;
-   unit 0 {
-       family bridge {
-           interface-mode trunk;
-           vlan-id-list 1-999;
-       }
-   }
-   unit 1 {
-       vlan-id 1000;
-       family inet {
-           address 10.8.0.0/31;
-       }
-       family iso;
-   }
+   hierarchical-scheduler;
+   vlan-tagging;
+   unit 0 {
+       family bridge {
+           interface-mode trunk;
+           vlan-id-list 1-999;
+       }
+   }
+   unit 1 {
+       vlan-id 1000;
+       family inet {
+           address 10.8.0.0/31;
+       }
+       family iso;
+   }
```

Now in H-CoS mode, the same error is generated, as nothing changes here regarding the interface's G-Rate booking:

```
NPC2(R1-RE0 vty)# [May 23 15:45:30.570 LOG: Warning] ifd(xe-2/0/0)
guaranteed_bw_remain (0) is less than ifl-334's configured guaranteed rate (2000000)
```

But now, all IFLs are shown with a G-Rate, to include IFL 1, and the sum of G-Rate booking is now 12 Mbps on a 10 Mbps link:

```
NPC2(R1-RE0 vty)# sho cos scheduler-hierarchy

class-of-service EGRESS scheduler hierarchy - rates in kbps
---------------------------------------------------------------------------------
                                    shaping guarntd delaybf  excess
interface name               index    rate    rate    rate    rate      other
---------------------------- -----  ------- ------- ------- ------- -------------
xe-2/0/0                       148    10000       0       0       0
  xe-2/0/0.0                   332    10000    5000       0       0
    q 0 - pri 0/0            20205        0      5%       0     35%
    q 1 - pri 0/0            20205        0      5%       0      5%
    q 2 - pri 0/0            20205        0     10%       0     10%
    q 3 - pri 3/1            20205        0      5%     10%      5%
    q 4 - pri 0/0            20205        0     30%       0     30%
    q 5 - pri 4/0            20205        0     30%   25000      0%     exact
    q 6 - pri 0/0            20205        0     15%       0     15%
    q 7 - pri 2/5            20205       0       0       0      0%
  xe-2/0/0.1                   333   10000    5000       0       0
    q 0 - pri 0/0            20205       0      5%       0     35%
    q 1 - pri 0/0            20205       0      5%       0      5%
    q 2 - pri 0/0            20205       0     10%       0     10%
    q 3 - pri 3/1            20205       0      5%     10%      5%
    q 4 - pri 0/0            20205       0     30%       0     30%
    q 5 - pri 4/0            20205       0     30%   25000      0%      exact
    q 6 - pri 0/0            20205       0     15%       0     15%
    q 7 - pri 2/5            20205       0       0       0      0%
  xe-2/0/0.32767               334       0    2000    2000       0
    q 0 - pri 0/1                2       0     95%     95%      0%
    q 3 - pri 0/1                2       0      5%      5%      0%
  xe-2/0/0-rtp                 148   10000       0       0       0
. . .
```

Another option is to increase the shaping rate of the IFD by 2 Mbps. The IFLs would still be limited to their 10 Mbps shaping rate, but now the extra 2 Mbps available at the IFD can be used when both are active at the same time, assuming there is no LACP traffic flowing, of course. If the shaped rate of the IFD has to stay at 10 Mbps, the remaining option is to reduce the G-Rate of each IFL by 1 Mbps, again to accommodate the 2 Mbps control scheduler, as its bandwidth is not configurable. In this case, the second choice is taken, namely remain in per unit and increase IFD shape rate:

```
{master}[edit]
jnpr@R1-RE0# show | compare
[edit class-of-service traffic-control-profiles tc_ifd_10m]
-    shaping-rate 10m;
+    shaping-rate 12m;
```

After committing the change, the new IFD shaped rate was found, as expected, and the error message was no longer seen. But, frustratingly, IFL 1 *still* did not get the configured G-Rate:

```
NPC2(R1-RE0 vty)# sho cos scheduler-hierarchy

class-of-service EGRESS scheduler hierarchy - rates in kbps
---------------------------------------------------------------------------------
                                    shaping guarntd delaybf  excess
interface name               index    rate    rate    rate    rate      other
---------------------------- -----  ------- ------- ------- ------- -------------
xe-2/0/0                       148   12000       0       0       0
  xe-2/0/0.0                   332   10000    5000       0       0
    q 0 - pri 0/0            20205       0      5%       0     35%
    q 1 - pri 0/0            20205       0      5%       0      5%
    q 2 - pri 0/0            20205       0     10%       0     10%
    q 3 - pri 3/1            20205       0      5%     10%      5%
    q 4 - pri 0/0            20205       0     30%       0     30%
    q 5 - pri 4/0            20205       0     30%   25000      0%      exact
    q 6 - pri 0/0            20205       0     15%       0     15%
    q 7 - pri 2/5            20205       0       0       0      0%
  xe-2/0/0.1                   333   10000       0       0       0
    q 0 - pri 0/0            20205       0      5%       0     35%
    q 1 - pri 0/0            20205       0      5%       0      5%
```

```
      q 2 - pri 0/0           20205       0     10%       0     10%
      q 3 - pri 3/1           20205       0      5%     10%      5%
      q 4 - pri 0/0           20205       0     30%       0     30%
      q 5 - pri 4/0           20205       0     30%   25000      0%      exact
      q 6 - pri 0/0           20205       0     15%       0     15%
      q 7 - pri 2/5           20205       0       0       0      0%
    xe-2/0/0.32767              334       0    2000    2000       0
      q 0 - pri 0/1               2       0     95%     95%      0%
      q 3 - pri 0/1               2       0      5%      5%      0%
  xe-2/0/1                      149       0       0       0       0
  xe-2/1/0                      150       0       0       0       0
  xe-2/1/1                      151       0       0       0       0
  xe-2/2/0                      152       0       0       0       0
  xe-2/2/1                      153       0       0       0       0
  xe-2/3/0                      154       0       0       0       0
  xe-2/3/1                      155       0       0       0       0
```

Even a `commit full` did not alter the state. As mentioned previously, there are cases where an interface itself had to be bounced, which is to say deactivated and reactivated, before a new G-Rate can be applied. Juniper development indicated this was working as designed, hence the advice to try flapping interfaces, or the whole CoS stanza, when recent changes seem to not take effect. Here, just the affected IFL is flapped to minimize any service disruption:

```
{master}[edit]
jnpr@R1-RE0# deactivate interfaces xe-2/0/0 unit 1

{master}[edit]
jnpr@R1-RE0# commit
re0:
configuration check succeeds
re1:
commit complete
re0:
commit complete

{master}[edit]
jnpr@R1-RE0# rollback 1
load complete

{master}[edit]
jnpr@R1-RE0# commit
re0:
configuration check succeeds
re1:
commit complete
re0:
commit complete
```

After the flap, all is as expected with both user IFLs getting their configured G-Rate:

```
NPC2(R1-RE0 vty)# sho cos scheduler-hierarchy

class-of-service EGRESS scheduler hierarchy - rates in kbps
---------------------------------------------------------------------------------
```

```
                                    shaping guarntd delaybf  excess
interface name               index    rate    rate    rate    rate      other
---------------------------- -----  ------- ------- ------- ------- -------------
xe-2/0/0                       148    12000       0       0       0
  xe-2/0/0.0                   332    10000    5000       0       0
    q 0 - pri 0/0            20205        0      5%       0     35%
    q 1 - pri 0/0            20205        0      5%       0      5%
    q 2 - pri 0/0            20205        0     10%       0     10%
    q 3 - pri 3/1            20205        0      5%     10%      5%
    q 4 - pri 0/0            20205        0     30%       0     30%
    q 5 - pri 4/0            20205        0     30%   25000      0%     exact
    q 6 - pri 0/0            20205        0     15%       0     15%
    q 7 - pri 2/5            20205        0       0       0      0%
  xe-2/0/0.1                   333    10000    5000       0       0
    q 0 - pri 0/0            20205        0      5%       0     35%
    q 1 - pri 0/0            20205        0      5%       0      5%
    q 2 - pri 0/0            20205        0     10%       0     10%
    q 3 - pri 3/1            20205        0      5%     10%      5%
    q 4 - pri 0/0            20205        0     30%       0     30%
    q 5 - pri 4/0            20205        0     30%   25000      0%     exact
    q 6 - pri 0/0            20205        0     15%       0     15%
    q 7 - pri 2/5            20205        0       0       0      0%
  xe-2/0/0.32767               334        0    2000    2000       0
    q 0 - pri 0/1                2        0     95%     95%      0%
    q 3 - pri 0/1                2        0      5%      5%      0%
```

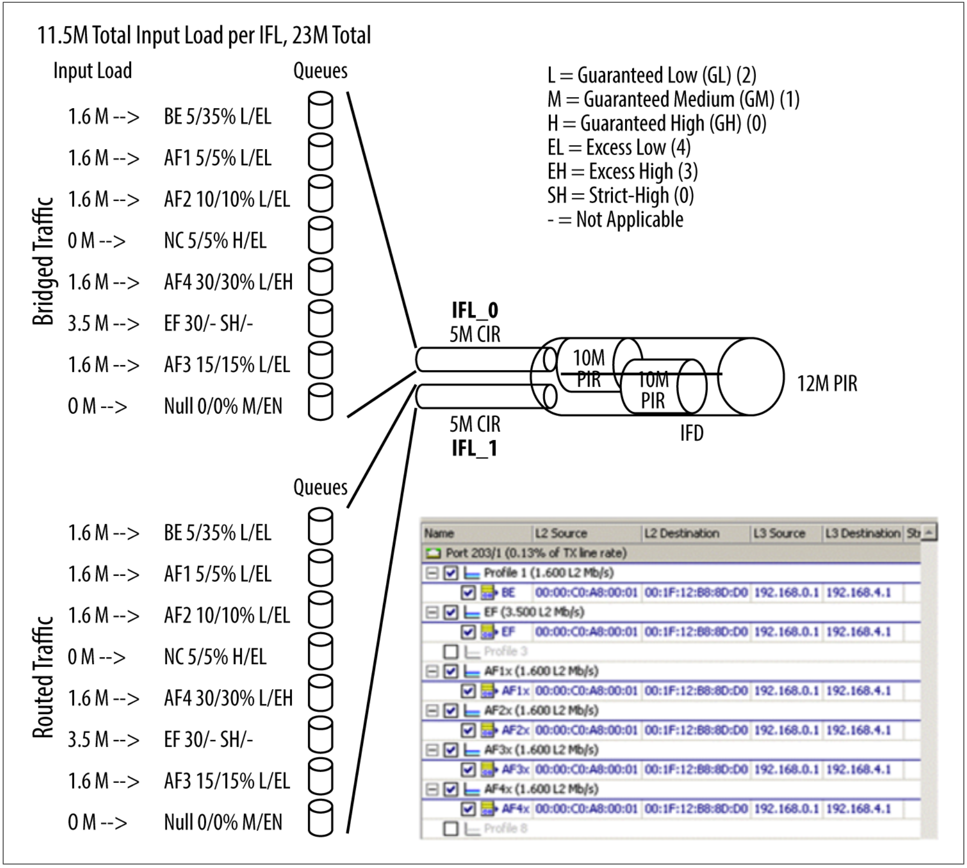
When CoS changes don't seem to be taking effect, it's a good idea to try a `commit full` (the full part is hidden). If that does not help, there are times where it's a good idea to deactivate the `class-of-service` stanza, then reactivate it to ensure all changes are placed into effect. There is a data plane impact to such actions, as a small burst of errors results from reconfigured queue buffers, etc., and during the flap the box will be using default CoS, so don't dally around.

Flapping just the affected interfaces, as shown in this section, is also known to activate CoS changes that are otherwise not taking effect.

Confirm Scheduling Behavior

It's very difficult to test CoS scheduling behavior using only ping. While useful for the basic classification and connectivity checks that brought us here, pings simply cannot generate enough traffic to cause congestion. Enter the Agilent Router Tester (ART), which allows both control and data plane stimulation. In this case, the traffic generator is used to exercise the data plane as that's the focus of CoS, which is control plane independent. In this topology, passive IS-IS is used to provide reachability to the L3-related content networks, and all traffic is generated/received from the direct subnet associated with the tester ports.

Before starting up the traffic, some predictions are in order. Recall that the IFD is shaped to 10 Mbps, as is each of the two IFLs. The IFLs are each given 5 Mbps of guaranteed bandwidth. Figure 5-30 shows the state of affairs for the two IFLS and their shared interface.



Name	L2 Source	L2 Destination	L3 Source	L3 Destination	St
Port 203/1 (0.13% of TX line rate)					
⊟ ☑ Profile 1 (1.600 L2 Mb/s)					
☑ BE	00:00:C0:A8:00:01	00:1F:12:B8:8D:D0	192.168.0.1	192.168.4.1	
⊟ ☑ EF (3.500 L2 Mb/s)					
☑ EF	00:00:C0:A8:00:01	00:1F:12:B8:8D:D0	192.168.0.1	192.168.4.1	
□ Profile 3					
⊟ ☑ AF1x (1.600 L2 Mb/s)					
☑ AF1x	00:00:C0:A8:00:01	00:1F:12:B8:8D:D0	192.168.0.1	192.168.4.1	
⊟ ☑ AF2x (1.600 L2 Mb/s)					
☑ AF2x	00:00:C0:A8:00:01	00:1F:12:B8:8D:D0	192.168.0.1	192.168.4.1	
⊟ ☑ AF3x (1.600 L2 Mb/s)					
☑ AF3x	00:00:C0:A8:00:01	00:1F:12:B8:8D:D0	192.168.0.1	192.168.4.1	
⊟ ☑ AF4x (1.600 L2 Mb/s)					
☑ AF4x	00:00:C0:A8:00:01	00:1F:12:B8:8D:D0	192.168.0.1	192.168.4.1	
□ Profile 8					

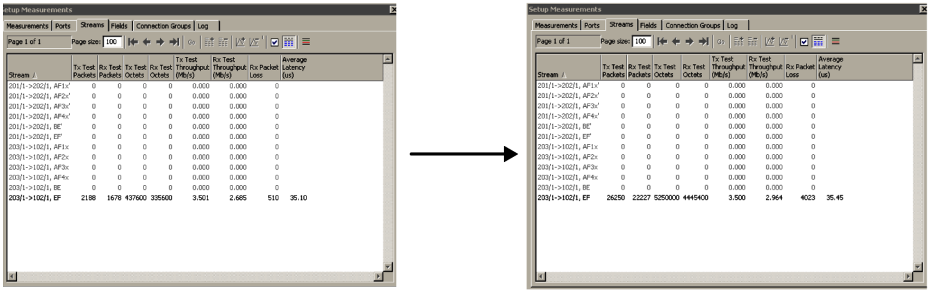
Figure 5-30. Per Unit Scheduling for Bridged and Routed Traffic.

Figure 5-30 highlights the IFD shaping rate of 12 Mbps and the two IFLs, each shaped to 10 Mbps, and each with a CIR/G-Rate of 5 Mbps. The two sets of queues, one for each IFL, are also shown, along with each queue/FCs, transmit rate/excess rate percentages, and their normal and excess rate scheduling priorities. The EF has no excess rate due to its use of rate limiting, and the Null class is blocked from using any excess. Most queues are using the default L/GL priority, and most have had their excess priority values altered from the defaults that are based on transmit rate. All queues eligible for excess have an explicit excess rate defined.

The easiest prediction one can level against this setup is that EF will always get its 3 Mbps first, due to its SH setting. As a basic sanity test, EF traffic is started on the L3 source. Things seem to go wrong from the start when the tester displays only 2.6 Mbps of traffic is received. With only the one source active, it's hard to believe there is any congestion of contention issues. Could there be some mismatch in data rate?

Match Tester's Layer 2 Rate to Trio Layer 1 Shaping

The Agilent router tester used in this lab does not provide an option to generate Ethernet traffic based on physical layer rate. In contrast, Trio chipsets display queue statistics and shape based on Layer 1 overhead, which for Ethernet includes the 96-bit time (12 bytes) interface gap as well as the 8-byte preamble, overhead that totals 20 bytes. At the frame layer, the tester and Trio both count Ethernet overhead that includes destination and source MACs (12 bytes), the Type/Length (2 bytes) field, and the 4-byte FCS. Given the incompatibility is at Layer 1, you can use the overhead accounting feature to subtract the 20 bytes of Layer 1 overhead from Trio accounting and shaping to match their rates, a real benefit when doing this type of testing. Figure 5-31 shows the before and after effects of matching the tester's traffic rate to the router's shaping function.



Stream	Tx Test Packets	Rx Test Packets	Tx Test Octets	Rx Test Octets	Tx Test Throughput (Mb/s)	Rx Test Throughput (Mb/s)	Rx Packet Loss	Average Latency (us)
201/1->202/1, AF1x'	0	0	0	0	0.000	0.000	0	
201/1->202/1, AF2x'	0	0	0	0	0.000	0.000	0	
201/1->202/1, AF3x'	0	0	0	0	0.000	0.000	0	
201/1->202/1, AF4x'	0	0	0	0	0.000	0.000	0	
201/1->202/1, BE'	0	0	0	0	0.000	0.000	0	
201/1->202/1, EF'	0	0	0	0	0.000	0.000	0	
203/1->102/1, AF1x	0	0	0	0	0.000	0.000	0	
203/1->102/1, AF2x	0	0	0	0	0.000	0.000	0	
203/1->102/1, AF3x	0	0	0	0	0.000	0.000	0	
203/1->102/1, AF4x	0	0	0	0	0.000	0.000	0	
203/1->102/1, BE	0	0	0	0	0.000	0.000	0	
203/1->102/1, EF	2188	1678	437600	335600	3.501	2.685	510	35.10

Stream	Tx Test Packets	Rx Test Packets	Tx Test Octets	Rx Test Octets	Tx Test Throughput (Mb/s)	Rx Test Throughput (Mb/s)	Rx Packet Loss	Average Latency (us)
201/1->202/1, AF1x'	0	0	0	0	0.000	0.000	0	
201/1->202/1, AF2x'	0	0	0	0	0.000	0.000	0	
201/1->202/1, AF3x'	0	0	0	0	0.000	0.000	0	
201/1->202/1, AF4x'	0	0	0	0	0.000	0.000	0	
201/1->202/1, BE'	0	0	0	0	0.000	0.000	0	
201/1->202/1, EF'	0	0	0	0	0.000	0.000	0	
203/1->102/1, AF1x	0	0	0	0	0.000	0.000	0	
203/1->102/1, AF2x	0	0	0	0	0.000	0.000	0	
203/1->102/1, AF3x	0	0	0	0	0.000	0.000	0	
203/1->102/1, AF4x	0	0	0	0	0.000	0.000	0	
203/1->102/1, BE	0	0	0	0	0.000	0.000	0	
203/1->102/1, EF	26250	22227	5250000	4445400	3.500	2.964	4023	35.45

```
{master}[edit]
jnpr@R1-RE0# set class-of-service traffic-control-profiles tc_ifd_10m
overhead-accounting bytes -20


{master}[edit]
jnpr@R1-RE0# set class-of-service traffic-control-profiles tc_l3_ifl_5m
overhead-accounting bytes -20


{master}[edit]
jnpr@R1-RE0# set class-of-service traffic-control-profiles tc_l2_ifl_5m
overhead-accounting bytes -20

{master}[edit]
jnpr@R1-RE0# commit
. . .
```

Figure 5-31. Adjust Overhead Accounting Bytes to Match Tester and DUT.

Note that the overhead accounting needs to be adjusted for all shapers in the chain in order to achieve the expected results. In this case, that means the two IFL shapers and the IFD-level shaper. If H-CoS were in play, you would also need to adjust any IFL-Set-level shapers. It's generally not a good idea to complicate things by mismatching the overhead accounting values at different scheduling nodes for a given interface, but such configurations are permitted.

With the overhead factors matched, the EF class shows the expected 3 Mbps of throughput, and loss, again expected given the input rate exceeds the rate limiting that's in effect for this queue. Note the end-to-end latency, which at 35 microseconds is rather low. This is confirmation of the lack of delay buffer/shaping for this queue. Pretty remarkable, given that is the delay through all three MX routers with a 10 Mbps link in the mix.

Compute Queue Throughput: L3

Looking at the queue priorities, transmit and excess rates, along with the IFL/IFD G-Rates and shaping rate, respectively, can you predict what will happen if all six traffic flows are started to generate the 11.5 Mbps shown? The same approach taken in the previous Pop Quiz section is used again here. After all, the approach worked there; if it's not broken, why fix it?

Recall that in the CoS PoC lab, static routing was used so there was no control plane traffic to muck up the results. To add realism, the current test network is running various Layer 3 routing and Layer 2 switching/control protocols, specifically IS-IS, BFD, STP, and the VRRP protocols. The tester is not configured to generate any NC traffic to avoid causing any problems with these protocols should NC congestion result. Interface monitoring at R1 in steady state indicated approximately 5 kbps of NC traffic is flowing in the background. The NC load is low enough that it can be safely disregarded as far as the calculations go.

The queue throughput calculations start with elimination of any low-hanging fruit. The Null queue is not being used, and the 5 kbps of background NC can safely be ignored. That brings us from eight to six queues that are of concern. Keep these tips in mind when thinking about the throughput for these queues:

The interface is in CIR/PIR mode due to a G-Rate being specified.

As before, it's best to start with any SH traffic to get it out of the way, and then work your way down based on priority.

Once all the G-Rates of GH/GM are met, you can determine if any CIR remains. If so, calculate GL queue bandwidth based on transmit rate ratio until you enter excess. Recall that GL gets demoted if the sum of GH/GM exceeds the G-Rate.

We're saying this again: GH and GM will get their G-Rate even if it has to come from the excess region.

When in the PIR/excess range, make sure to factor excess priority. Remember that EH will starve EL in the excess if there are no rate limits at work. For queues at the same excess priority, the sharing is based on the ratio of their excess rates.

A previous PFE display indicated that based on CIR, the non-EF queues guaranteed-rates are Q0/Q1/Q3: 250 kbps; Q2: 500 kbps; Q4: 1.5 Mbps; Q5: NA; and Q6: 750 kbs. The sum of non-EF G-Rates is therefore 3 Mbps. Adding the 3 Mbps of EF brings this to 6 Mbps, and the IFL has a CIR of 5 Mbps. Clearly, not all queues can get their CIR in the guaranteed region.

There is a difference between a queue's maximum and actual rate. Previous examples had all queues overdriven, such that maximum rate was achieved. Here, the input rate of 1.6 Mbps is less than the maximum rate of some queues, thereby allowing others to make use of their leftover capacity.

The Layer 3 IFL Calculation: Maximum. While the primary goal is actual queue throughput, both the maximum and actual queue rates are computed for the sake of completeness. The calculation proceeds according to the previous guidelines. First, compute the maximum rates when all queues are overdriven (which is *not* the current case):

Guaranteed Region:

5 Mbps CIR/5 Mbps PIR

Q5/EF gets 3 Mbps,

2 Mbps CIR/5 Mbps PIR remains

(Q3/NC at GH skipped, BG protocol traffic ~ 5 kbps)
(Q7/Null at GM skipped no input)

Five queues remain, all at GL, sharing G-Rate based on TX ratio; the sum of the five queues' transmit weights equal 65 (5 + 5 + 10 + 30 + 15 = 65). That number forms the basis of the ratio calculation, and recall that 2 Mbps of CIR remains:

Q0/BE: 2 * 5/65 (0.076) = 0.152 Mbps
Q1/AF1: 2 * 5/65 (0.076) = 0.152 Mbps
Q2/AF2: 2 * 10/65 (0.153) = 0.306 Mbps
Q4/AF4: 2 * 30/65 (0.461) = 0.922 Mbps
Q6/AF3: 2 * 15/65 (0.230) = 0.460 Mbps
Total: 3 Mbps + 1.99 Mbps = 4.99 Mbps, 0 Mbps CIR/5 Mbps PIR remains

Excess Region: 5 Mbps PIR

With all G-Rate consumed, the L3 scheduler node demotes the queues. They are all at GL and so demotable, and therefore despite some not having reached their configured transmit rate, into the excess region (PIR) they all go with 5 Mbps of PIR to fight over.

The sum of the five queues' excess rates equals 95 (35 + 5 + 10 + 30 + 15 = 95). That number forms the basis of the ratio calculation, and recall that 5 Mbps of PIR remains. Note that here percentages are being used. The same results can be had using the proportional weights shown in the previous `sho cos halp ifl 332` command output, where the weights are scaled by a factor of 10 to sum to 1,000, making Q0's 35% into 350 while Q1's 5% is 50, etc.

Q0/BE: 5 * 35/95 (0.368) = 1.84 Mbps
Q1/AF1: 5 * 5/95 (0.052) = 0.26 Mbps
Q2/AF2: 5 * 10/95 (0.105) = 0.525 Mbps
Q4/AF4: 5 * 30/95 (0.315) = 1.57 Mbps
Q6/AF3: 5 * 15/95 (0.157) = 0.785 Mbps
Total = 4.99 Mbps

Queue maximums when overdriven:

Q0: 0.152 Mbps + 1.84 Mbps = 1.992 Mbps

Q1: 0.152 Mbps + 0.26 Mbps = 0.412 Mbps

Q2: 0.306 Mbps + 0.525 Mbps = 0.831 Mbps

Q3: ~ 5 kbps

Q4: 0.922 Mbps + 1.57 Mbps = 2.492 Mbps

Q5: 3 Mbps

Q6: 0.461 Mbps + 0.785 Mbps = 1.256 Mbps

Total: 9.983 M

The Layer 3 IFL Calculation: Actual Throughput. The calculation for actual queue throughput with the input loads shown begins with the same steps as the previous maximum throughput example. Things change when performing the excess rate calculations, however.

From the previous calculation G-Rate bandwidth was allocated as shown:

Q0/BE: 2 * 5/65 (0.076) = 0.152 Mbps

Q1/AF1: 2 * 5/65 (0.076) = 0.152 Mbps

Q2/AF2: 2 * 10/65 (0.153) = 0.306 Mbps

Q4/AF4: 2 * 30/65 (0.461) = 0.922 Mbps

Q6/AF3: 2 * 15/65 (0.230) = 0.460 Mbps

Total: 3 M + 1.99 Mbps = 4.99 M, 0M CIR/5 Mbps PIR remains

Excess region: 5 Mbps PIR, total load 8 Mbps

We approach the excess sharing calculation a bit differently in this example. Given there are now five queues with widely varying rates, and that the input load is less than some queue's maximum rate, it makes sense to order the queues from highest to lowest excess rate and then see how each makes out. There is a total of 95% excess bandwidth share among the five queues, and 5 Mbps to share. In rough numbers, this works out to approximately 19% per 1 Mbps of excess bandwidth.

Taking Q0 as an example, it was left with 0.152 Mbps of G-Rate bandwidth. It has an input load of 1.6 M, making the difference 1.448 Mbps.With the excess-to-bandwidth ratio in effect, that means Q0 needs 27.5% excess bandwidth to satisfy its load, leaving 7.5% of its excess weight unused. The process is repeated next on Q4, as it has the highest remaining excess share, which is 30%.

The excess usage is computed. The percentages shown reflect excess bandwidth used, not the excess rate values that are configured. Some classes are satisfied before they use their full percentage, allowing other queues to borrow their unused excess.

Q0: 1.448 Mbps (27.5%)

Q1: 566 kbps (10.7%),

Q2: 1.13 Mbps (21.4%)

Q3: ~ 5 kbps

Q4: 0.678 Mbps (12.8%)

Q5: 0 Mbps/rate limited

Q6: 0.114 Mbps (21.6%)

Q7: 0 Mbps/excess none

Total: 4.9 Mbps

Table 5-14 summarizes the CIR and PIR rate calculation results for all queues.

Table 5-14. Excess Region Bandwidth Allocation and Queue Total Usage (in Mbps).

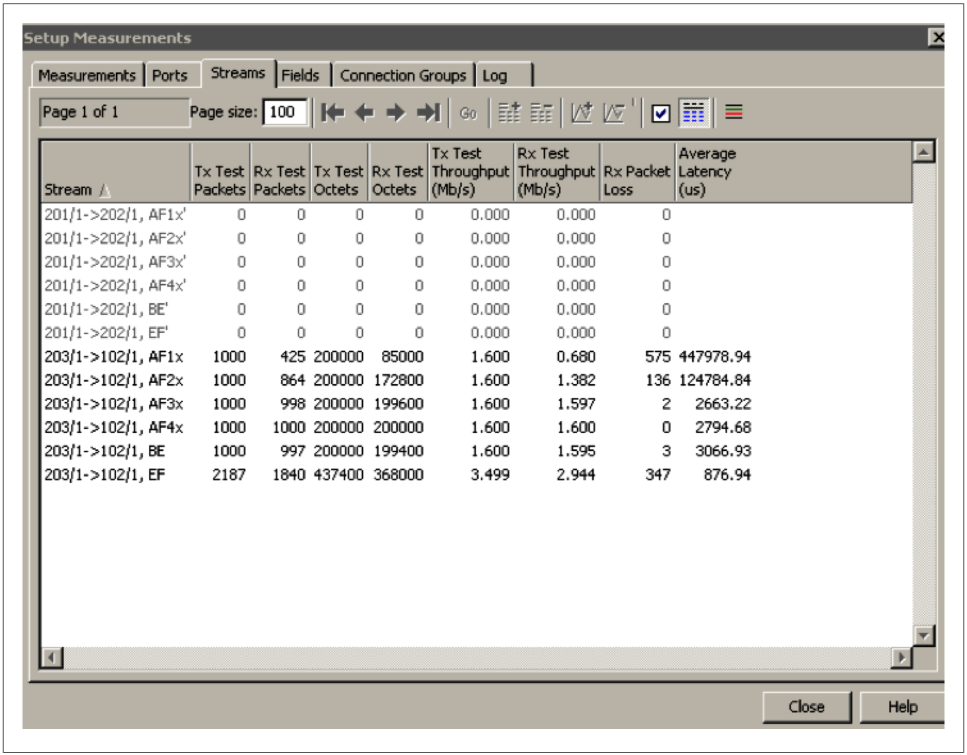
Queue	Load	Excess rate	G-Rate BW	Needs	Gets	Excess Rate Used	Excess Rate +/ –	PIR Left (5 m)	Queue Total (CIR + PIR)
0/BE	1.6 M	35%	0.152 M	1.44 M	1.44 M	27.5%	–7.5	3.55 M	0.152 + 1.448 = 1.6 (0 loss)
4/AF4	1.6 M	30%	0.922 M	0.678 M	0.678 M	12.8%	–17.2	2.87 M	0.922 + 0.678 = 1.6 (0 loss)
6/AF3	1.6 M	15%	0.460 M	1.14 M	1.14 M	21.6%	+6.6	1.73 M	0.460+1.14=1.6 (0 loss)
2/AF2	1.6 M	10%	0.306 M	1.29 M	1.13 M	21.4%	+11	570 K	0.306 + 1.13 = 1.4M (144PPS loss
1/AF1	1.6 M	5%	0.152 M	1.448 M	566 K	10.7%	+5	0	0.152 + 0.566 = 0.71M (500 PPS loss)

As shown, queues 1, 4, and 6 can support the offered load, with the first two queues having excess capacity to spare; the excess represents additional load that queue could carry. As a result, no loss is expected for these queues. In the case of queue 0, the 1.6

Mbps shown plus excess bandwidth represented by its remaining 7.5 would bring the queue to its maximum 1.9 Mbps computed previously.

When we get to queues 2 and 1, which are at a 2:1 sharing ratio, there is only 1.7 Mbps PIR remaining. Clearly, both queues will not have their CIRs met. Given the ratio, queue 2 gets its two-thirds and queue 1 gets the remaining one-third.

Figure 5-32 shows the measured result when only the Layer 3 flows are active.



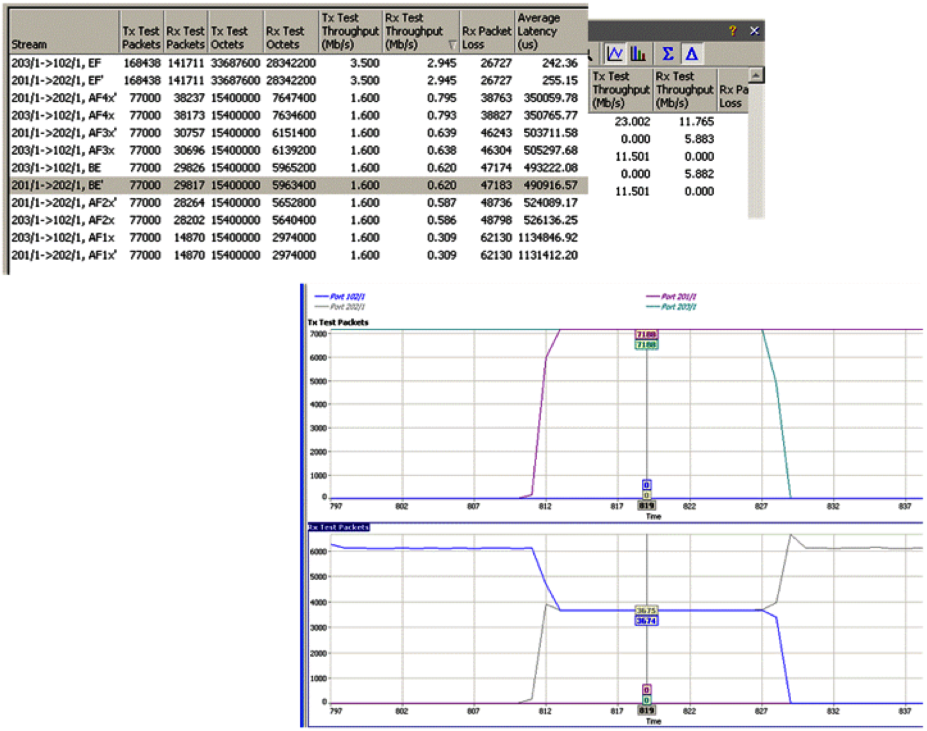
Stream	Tx Test Packets	Rx Test Packets	Tx Test Octets	Rx Test Octets	Tx Test Throughput (Mb/s)	Rx Test Throughput (Mb/s)	Rx Packet Loss	Average Latency (us)
201/1->202/1, AF1x'	0	0	0	0	0.000	0.000	0	
201/1->202/1, AF2x'	0	0	0	0	0.000	0.000	0	
201/1->202/1, AF3x'	0	0	0	0	0.000	0.000	0	
201/1->202/1, AF4x'	0	0	0	0	0.000	0.000	0	
201/1->202/1, BE'	0	0	0	0	0.000	0.000	0	
201/1->202/1, EF'	0	0	0	0	0.000	0.000	0	
203/1->102/1, AF1x	1000	425	200000	85000	1.600	0.680	575	447978.94
203/1->102/1, AF2x	1000	864	200000	172800	1.600	1.382	136	124784.84
203/1->102/1, AF3x	1000	998	200000	199600	1.600	1.597	2	2663.22
203/1->102/1, AF4x	1000	1000	200000	200000	1.600	1.600	0	2794.68
203/1->102/1, BE	1000	997	200000	199400	1.600	1.595	3	3066.93
203/1->102/1, EF	2187	1840	437400	368000	3.499	2.944	347	876.94

Figure 5-32. CoS Lab Measured Results: L3 Flow.

Once again, the measured results correlate well with the computed values. As a final pop quiz, ask yourself what will happen to traffic on IFL 1's queues if the L2 test streams are stated up on IFL 0. Given that both IFLs have a peak rate of 10 Mbps due to shaping, it's clear that both IFLs can never send at their peak rate simultaneously. One IFL or the other can burst to the IFL's shaped speed of 10 M, but this leaves only 2 Mbps of IFD-shaped bandwidth for the other IFL (recall IFD is shaped to 12 Mbps to accommodate the LACP control scheduler, but it's not sending any traffic as LACP is not enabled).

The most you can ever hope to get over the IFD is 12 Mbps, and the traffic on the two IFLs can reach an aggregate rate of 23 Mbps, so something has to hit the floor. With rough math, it seems that each IFL can expect its 5 Mbps CIR + a share of the 2 Mbps PIR that remains. This means you expect each queue to lose much of its excess bandwidth and fall back to the CIR region. Therefore, both EF queues will remain at 3 Mbps while BE should drop to some 0.6 Mbps, almost as high as AF4x. This may surprise you, but it makes sense when you consider that AF4x gets most of the 2 Mbps CIR (30%) while both AF4x and BE gets about one-third of the 1 Mbps PIR, thus keeping AF4x in the lead over BE, with both coming in behind EF.

Figure 5-33 shows the measured results when both IFLs are driven at the same time and at the same rates.



Stream	Tx Test Packets	Rx Test Packets	Tx Test Octets	Rx Test Octets	Tx Test Throughput (Mb/s)	Rx Test Throughput (Mb/s)	Rx Packet Loss	Average Latency (us)
203/1->102/1, EF	168438	141711	33687600	28342200	3.500	2.945	26727	242.36
201/1->202/1, EF'	168438	141711	33687600	28342200	3.500	2.945	26727	255.15
201/1->202/1, AF4x'	77000	38237	15400000	7647400	1.600	0.795	38763	350059.78
203/1->102/1, AF4x	77000	38173	15400000	7634600	1.600	0.793	38827	350765.77
201/1->202/1, AF3x'	77000	30757	15400000	6151400	1.600	0.639	46243	503711.58
203/1->102/1, AF3x	77000	30696	15400000	6139200	1.600	0.638	46304	505297.68
203/1->102/1, BE	77000	29826	15400000	5965200	1.600	0.620	47174	493222.08
201/1->202/1, BE'	77000	29817	15400000	5963400	1.600	0.620	47183	490916.57
201/1->202/1, AF2x'	77000	28264	15400000	5652800	1.600	0.587	48736	524089.17
203/1->102/1, AF2x	77000	28202	15400000	5640400	1.600	0.586	48798	526136.25
203/1->102/1, AF1x	77000	14870	15400000	2974000	1.600	0.309	62130	1134846.92
201/1->202/1, AF1x'	77000	14870	15400000	2974000	1.600	0.309	62130	1131412.20

Tx Test Throughput (Mb/s)	Rx Test Throughput (Mb/s)	Rx Pa Loss
23.002	11.765	
0.000	5.883	
11.501	0.000	
0.000	5.882	
11.501	0.000	

Figure 5-33. CoS Lab Measured Results: Both Flows Active.

The chart in the upper left is at steady state with all streams flowing. The WRED profile and chronic congestion that was not present in the single IFL case do skew the numbers a bit, but they are symmetrical between the queues on both IFLs, proving neither is getting any advantage, as is expected given their identical settings. Note that EF is unaffected given its SH priority and LLQ settings that result in a small buffer mean that congestion does not affect this queue as much as the others, at least from a latency perspective. The line graph on the right shows a sequence with just the Layer 3 IFL sending, then both IFLs, and then just the Layer 2. Again, matched rates are confirmed during the period when both are active, and it's clear that when one IFL is inactive, the other can take advantage by moving into the PIR region for excess bandwidth. In this setup, one IFL's CIR is the other's PIR, a fact reflected nicely in the graph. The upper right confirms the total offers load of 23 Mbps versus the total received load, which hovers nicely around the 12 Mbps IFD shaping rate.

The math to compute the BE queues follows, if you are curious. Again, NC and Null are left out:

The sum of the transmit rates for the five queues equals 65 (5 + 5 + 10 + 30 + 15 = 65), while the sum of queue excess rate equals 85 (35 + 5 + 10 + 30 + 5 = 85). These numbers form the basis of the ratio calculation for both CIR and PIR sharing respectively; the ratios are converted to decimal:

CIR sharing:

5/65 = 0.076

10/65 = 0.153

15/65 = 0.230

30/65 = 0.461

PIR sharing:

5/85 = 0.058

10/85 = 0.117

30/85 = 0.352

35/85 = 0.411

And now the math, which is the same for both IFLs:

5 Mbps CIR/1 Mbps PIR

EF gets 3 Mbps, 2 Mbps CIR/1 Mbps PIR remains

CIR Bandwidth (2 Mbps available for all five queues):

Q0/BE: 2 * 0.076 = 0.152 Mbps

Q1/AF1: 2 * 0.076 = 0.152 Mbps

Q2/AF2: 2 * 0.153 = 0.306 Mbps

Q4/AF4: 2 * 0.461 = 0.922 Mbps

Q6/AF3: 2 * 0.230 = 0.460Mbps

Total = 1.992 Mbps, 0 Mbps CIR, 1 Mbps PIR remains

Excess Bandwidth (1 Mbps available for all five queues):

Q0/BE: 1 * 0.461 = 0.461 Mbps

Q1/AF1: 1 * 0.058 = 0.058 Mbps

Q2/AF2: 1 * 0.117 = 0.117 Mbps

Q4/AF4: 1 * 0.352 = 0.352 Mbps

Q6/AF3: 1 * 0.058 = 0.058 Mbps

Total = 1.0 Mbps, 0 Mbps CIR, 0 Mbps PIR.

Queue totals, both IFLs active:

Q0/BE: 0.152 Mbps + 0.461 Mbps = 0.613 Mbps

Q1/AF1: 0.152 Mbps + 0.058 Mbps = 0.210 Mbps

Q2/AF2: 0.306 Mbps + 0.117 Mbps = 0.423 Mbps

Q4/AF4: 0.922 Mbps + 0.352 Mbps = 1.27 Mbps

Q5/EF: (all from CIR) = 3 Mbps

Q6/AF3: 0.460 Mbps + 0.058 Mbps = 0.518 Mbps

Total: 6.03 Mbps (5 Mbps CIR + 1 Mbps PIR)

Again, the numbers are remarkably close to the measured results. AF4x is the exception here, showing a lower than computed result. It's suspected that presence of the background NC, which is not taken into account in these calculations, is forcing demotion into excess (NC is high-priority and so always gets its G-Rate) before it has received its 30% share of the 2 Mbps CIR. Once in excess, it starts losing to BE with its higher excess rate. This is one of the many mysteries of CoS that keeps the job fun.

Before wrapping this up, the ultimate proof in the CoS pudding comes with pings generated from S1 to the L3 receiver using different ToS values, as these best simulate a Internet user's experience and how it can vary based on what class they are assigned. Note these tests are conducted when both traffic flows are present to induce chronic congestion as described in the previous section. All queues except for Q3/NC and Q7/Null, which are not driven by the tester, are showing packet drops and all of the available (shaped) bandwidth is utilized on the egress link at R1. First, the BE experience:

```
{master:0}[edit]
jnpr@S1-RE0# run ping 192.168.4.1
PING 192.168.4.1 (192.168.4.1): 56 data bytes
64 bytes from 192.168.4.1: icmp_seq=1 ttl=62 time=446.136 ms
64 bytes from 192.168.4.1: icmp_seq=3 ttl=62 time=484.471 ms
64 bytes from 192.168.4.1: icmp_seq=4 ttl=62 time=431.219 ms
64 bytes from 192.168.4.1: icmp_seq=5 ttl=62 time=542.507 ms
64 bytes from 192.168.4.1: icmp_seq=7 ttl=62 time=426.771 ms
64 bytes from 192.168.4.1: icmp_seq=8 ttl=62 time=439.971 ms
64 bytes from 192.168.4.1: icmp_seq=9 ttl=62 time=596.102 ms
64 bytes from 192.168.4.1: icmp_seq=12 ttl=62 time=488.459 ms
^C
--- 192.168.4.1 ping statistics ---
```

```
14 packets transmitted, 8 packets received, 42% packet loss
round-trip min/avg/max/stddev = 426.771/481.954/596.102/56.358 ms
```

The BE class is showing significant loss, and look at that delay. Oh, the humanity! Still, based on its 5%/35% and default WRED profile, it's getting better treatment than AF12:

```
{master:0}[edit]
jnpr@S1-RE0# run ping 192.168.4.1 tos 48 rapid count 100
PING 192.168.4.1 (192.168.4.1): 56 data bytes
........................^C
--- 192.168.4.1 ping statistics ---
25 packets transmitted, 0 packets received, 100% packet loss

{master:0}[edit]
jnpr@S1-RE0#
```

Recall that the AF12 class is using a rather aggressive drop profile that is putting the proverbial hurt on its traffic. Next, let's try the EF class:

```
{master:0}[edit]
jnpr@S1-RE0# run ping 192.168.4.1 tos 184
PING 192.168.4.1 (192.168.4.1): 56 data bytes
64 bytes from 192.168.4.1: icmp_seq=0 ttl=63 time=468.405 ms
64 bytes from 192.168.4.1: icmp_seq=1 ttl=62 time=3.133 ms
64 bytes from 192.168.4.1: icmp_seq=2 ttl=62 time=1.341 ms
^C
--- 192.168.4.1 ping statistics ---
4 packets transmitted, 3 packets received, 25% packet loss
round-trip min/avg/max/stddev = 1.341/157.626/468.405/219.755 ms
```

The result is a bit ironic, showing that EF is just like first class; it's great, but only when you can get in. Recall this class is overdriven at ingress based on its rate limiter, and as such, it's fast for the traffic that is accepted, but otherwise it drops with the best of them. Given what you know of the NC scheduler's priority, and its lack of loading, it seems that acting like NC is the way to go for the best performance possible in the current congested state:

```
{master:0}[edit]
jnpr@S1-RE0# run ping 192.168.4.1 tos 225 count 3
PING 192.168.4.1 (192.168.4.1): 56 data bytes
64 bytes from 192.168.4.1: icmp_seq=0 ttl=62 time=1.476 ms
64 bytes from 192.168.4.1: icmp_seq=1 ttl=62 time=1.288 ms
64 bytes from 192.168.4.1: icmp_seq=2 ttl=62 time=4.055 ms

--- 192.168.4.1 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max/stddev = 1.288/2.273/4.055/1.262 ms
```

Not only is the NC queue low delay, but it's also loss free:

```
jnpr@S1-RE0# run ping 192.168.4.1 tos 225 rapid count 100
PING 192.168.4.1 (192.168.4.1): 56 data bytes
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
--- 192.168.4.1 ping statistics ---
```

```
100 packets transmitted, 100 packets received, 0% packet loss
round-trip min/avg/max/stddev = 1.001/1.832/6.850/1.279 ms
```

A final confirmation step displays interface queue statistics to confirm WRED versus rate limiter drops, starting with the EF queue:

```
{master}[edit]
jnpr@R1-RE0# run show interfaces queue xe-2/0/0 forwarding-class ef
Physical interface: xe-2/0/0, Enabled, Physical link is Up
  Interface index: 148, SNMP ifIndex: 4373
Forwarding classes: 16 supported, 8 in use
Egress queues: 8 supported, 8 in use
Queue: 5, Forwarding classes: ef
  Queued:
    Packets              :                36516                  1834 pps
    Bytes                :              7449264               2993600 bps
  Transmitted:
    Packets              :                36516                   1834 pps
    Bytes                :              7449264               2993600 bps
    Tail-dropped packets :                    0                     0 pps
    RL-dropped packets   :                 6946                   680 pps
    RL-dropped bytes     :              1416984               1110400 bps
    RED-dropped packets  :                    0                     0 pps
     Low                 :                    0                     0 pps
     Medium-low          :                    0                     0 pps
     Medium-high         :                    0                     0 pps
     High                :                    0                     0 pps
    RED-dropped bytes    :                    0                     0 bps
     Low                 :                    0                     0 bps
     Medium-low          :                    0                     0 bps
     Medium-high         :                    0                     0 bps
     High                :                    0                     0 bps
```

The EF queue confirms only rate limit (`RL-dropped`) drops, as expected. By design, this queue is always congestion free in order to keep latency low, trading low for delay every day of the week. Note that packets dropped due to rate limiting are never actually queued, and as such you don't expect rate limit drops to be reflected in packet queue or RED dropped statistics, as is the case here.

Meanwhile, the NC queue has had no drops, again in keeping with its high priority and lack of loading:

```
{master}[edit]
jnpr@R1-RE0# run show interfaces queue xe-2/0/0 forwarding-class nc
Physical interface: xe-2/0/0, Enabled, Physical link is Up
  Interface index: 148, SNMP ifIndex: 4373
Forwarding classes: 16 supported, 8 in use
Egress queues: 8 supported, 8 in use
Queue: 3, Forwarding classes: nc
  Queued:
    Packets              :                 1020                    10 pps
    Bytes                :                77999                  7632 bps
  Transmitted:
    Packets              :                 1020                     10 pps
    Bytes                :                77999                  7632 bps
    Tail-dropped packets :                      0                     0 pps
    RED-dropped packets  :                      0                     0 pps
     Low                 :                      0                     0 pps
     Medium-low          :                      0                     0 pps
     Medium-high         :                      0                     0 pps
     High                :                      0                     0 pps
    RED-dropped bytes    :                      0                     0 bps
     Low                 :                      0                     0 bps
     Medium-low          :                      0                     0 bps
     Medium-high         :                      0                     0 bps
     High                :                      0                     0 bps
```

In contrast, the AF1x class confirms historic and ongoing WRED drops:

```
{master}[edit]
jnpr@R1-RE0# run show interfaces queue xe-2/0/0 forwarding-class af1x
Physical interface: xe-2/0/0, Enabled, Physical link is Up
  Interface index: 148, SNMP ifIndex: 4373
Forwarding classes: 16 supported, 8 in use
Egress queues: 8 supported, 8 in use
Queue: 1, Forwarding classes: af1x
  Queued:
    Packets              :                   2181                  1998 pps
    Bytes                :                 444924               3265488 bps
  Transmitted:
    Packets              :                    407                    373 pps
    Bytes                :                  83028                609376 bps
    Tail-dropped packets :                      1                     0 pps
    RED-dropped packets  :                   1773                  1625 pps
```

The display confirms drops at a 1.6k PPS rate in the AF1x queue, a number that represents drops for all three AF1x classes. A previous measured result confirmed less drops in AF11 versus AF12, confirming that DiffServ design goals have been met. As expected, the level of drops is less in AF2, AF3, and AF4, respectively:

```
{master}[edit]
jnpr@R1-RE0#
run show interfaces queue xe-2/0/0 forwarding-class af2x | match RED-dropped
    RED-dropped packets  :                 280612                  1293 pps
    RED-dropped bytes    :               57244848               2113088 bps

{master}[edit]
jnpr@R1-RE0#
run show interfaces queue xe-2/0/0 forwarding-class af3x | match RED-dropped
    RED-dropped packets  :                 503119                  1214 pps
    RED-dropped bytes    :              102636276               1983144 bps

{master}[edit]
jnpr@R1-RE0#
run show interfaces queue xe-2/0/0 forwarding-class af4x | match RED-dropped
    RED-dropped packets  :                 426313                   956 pps
    RED-dropped bytes    :               86967852               1561104 bps
```

These results conclude the basic Trio CoS deployment lab.

Add H-CoS for Subscriber Access

People just can't seem to let a DiffServ network rest. Now that you have IP CoS up and running, the word is that you have to extend this CoS into a subscriber access network that is expected to scale beyond 1,000 users. The design has to be scalable (obviously) and has to offer not only IP DiffServ-based CoS, but also several performance profiles that are needed to enable triple-play services and to facilitate a tiered service offering.

In this example, different levels of performance are achieved through varying IFL shaping rates and level 2 scheduler node overbooking ratios. Clearly, users that are on IFLs with a higher shaping rate with little to no overbooking can expect better performance than those that have to contend for a overbooked group access rate while still being limited by a lower IFL speed. As noted previously, an alternative and safer design option is to only overbook PIR rates.

The design must offer per IFL/VLAN CoS profiles with eight queues per user. So at a minimum, per unit scheduling is needed. In addition, the network architects have asked that there be an interface-level usage limit on high-priority traffic. In like fashion, they also want each subscriber access class (business versus basic) to have an overall usage cap placed the same high-priority traffic. This high-priority traffic is special for having a high scheduling priority, and because it's often used to carry loss and delay sensitive real-time traffic as well as network control (albeit in separate queues). Even though each user IFL has a cap placed on total traffic via the IFL shaping rate, and there is a rate limit in effect for the EF queue, the concern is that "nothing fails like success" and the designers want overall caps placed on this traffic so that they can predict and model network performance without having to factor down to the individual subscriber/VLAN level. The hope is that with a hierarchical CoS design, the network planners can predict worse-case EF loads on a per access network basis without having to concern themselves with the day-to-day adds, moves, and changes in the subscriber network.

Because provisioning new users and executing move/change orders is a complicated process, configuration mistakes often occur. The design must therefore include a default set of queues at both the IFL-Set and IFD levels for users that, for whatever reason, either don't have an IFL-level TCP applied or don't belong to an official service tier (i.e., their IFLs are not listed in any known interface set). The H-CoS remaining construct fits this requirement nicely.

Defining a remaining profile at the IFD level, and for IFL-Sets, is a best practice when deploying H-CoS. As described previously, each remaining profile provides a set of shared queues that act as a safety net, catching users that are victims of provisioning errors, or possibly those attempting unauthorized access. While remaining profiles can provide a legitimate service class of their own, perhaps acting as the best-effort service container, it's also legitimate to provision remaining profiles so they give the attached users just enough service to be *miserable, so they complain*, possibly via email or even over the shiny new IP phone that came with their premium triple-play service. In the model, the complaints from authorized users get the provisioning mistakes corrected, and those that are attempting free service perhaps move on to a target with more bandwidth.

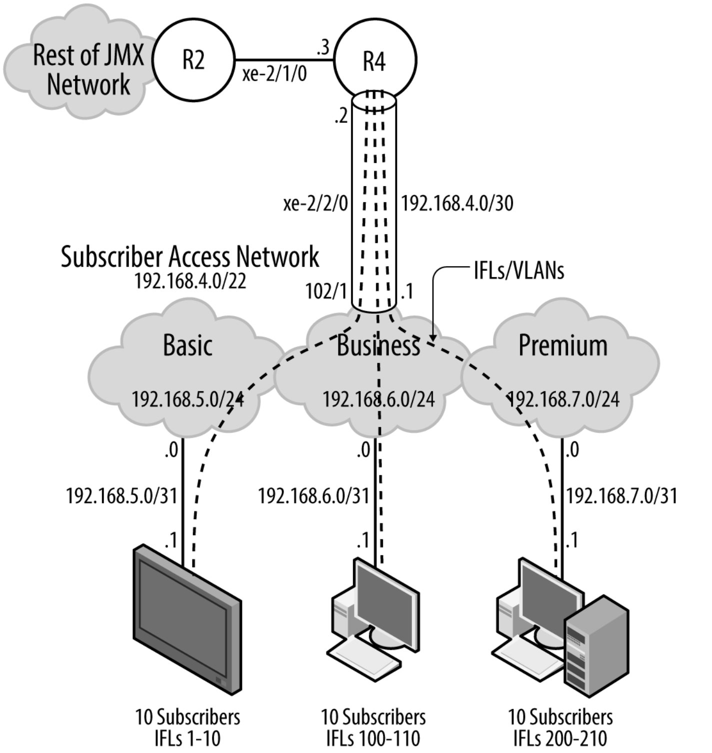
Even if you don't plan to use remaining traffic profiles (RTPs), defining them is best practice to help guard against unexpected results that can result from partial or misconfigurations.

Recall that IFLs that do not have a TCP attached at the `[edit class-of-service]` hierarchy, and which are not listed in any interface set, are supposed to be handled by the IFD's RTP, while IFLs that are listed in a set, but which also do not have a TCP attached, are supposed to be handled by the IFL-Set's RTP. Failing to define these RTPs can result in an IFL getting significantly more bandwidth than you intended. This is because when a specific RTP is not applied, the RTP profile inherits the node's shaping rate as a default. Thus for an IFD that is not part of a set, it gets the full IFD shaping rate while in the IFL named to a set case the IFL is expected to get the full IFL-Set's shaping rate. In both cases, a single IFL is allowed to consume shaped bandwidth that was likely intended for a group of IFLs, or in the case of the IFD's shaped rate, the bandwidth intended to be shared by a group of IFL-Sets!

PR 783690 was raised for an unconfigured IFL-Set remaining profile displaying the IFD shaping rate; the expected behavior, as described previously, is for the default IFL-Set RTP to display the IFL-Set's shaping rate. Despite this display issue, testing showed that such an IFL got the IFL-Set's PIR, as was expected, even though the IFD's PIR was displayed, making this issue appear cosmetic.

Defining remaining profiles, and attaching them to the IFD and to all interface sets, avoids both issues and is therefore the current best practice when deploying H-CoS. With explicit RTPs in place, any IFL that is listed in a set, but which does not have its own TCP applied, is attached to the IFL-Set's remaining profile, where it inherits the remaining profile's shaping and guaranteed rates in a predictable fashion. Likewise, any IFL that is not listed in an interface set, and which also does not have a TCP applied, is now attached to the IFD's remaining traffic profile.

With the basic design goals in place, it's time to get down to H-CoS configuration. The bulk of the CoS configuration from the previous section remains unchanged. In fact, all the changes happen at R4. Figure 5-34 shows the high-level CoS design.



H-CoS IFD Shaping Rates

IFD	500M PIR
GH Priority	200M PIR

H-CoS IFL and IFL_Set Shaping and G-Rates

Subscriber Type	IFL	IFL-Set	Per-Priority Shaping	Booking @ 10 users
Basic	1M PIR/500K CIR	2M PIR/1M CIR	GH 1M	5:l overbooked
Business	2M PIR/1M CIR	10M PIR/5M CIR	GH 1M	2:l overbooked
Premium	3M PIR/2M CIR	30M PIR/20M CIR	NA	1:1
Other	varies	1M PIR	500K	Varies, remaining IFL-Set for unprovisioned

Figure 5-34. H-CoS Lab Topology.

Before getting into H-CoS proper, let's cover a few things about the network design and IP addressing details shown in Figure 5-34. A 192.168.4/22 supernet is available to number the subscriber access network. The plan is to allocate a range of IFLs to each service tier, and these IFLs will each be associated with a matching VLAN tag and an IP address from the address pool assigned to the subscriber access network. The /22 aggregate route (supernet) yields a total of four/24 subnets; specifically, 192.168.4.0/24 through 192.168.7.0/24. In this design, subscriber access links use a /31 mask to allow up to 128 subscriber links per subnet/service class.

The basic service is expected to have approximately 100 users, provisioned on IFLs 1 to 99, using VLAN IDs 1 to 99, and assigned IP addresses from the 192.168.5/24 space. An aggregate route is defined at R4 and redistributed into IS-IS to accommodate routing into the subscriber network:

```
[edit]
jnpr@R4# show | compare rollback 1
[edit]
+  routing-options {
+      aggregate {
+          route 192.168.4.0/22;
+      }
+  }
[edit protocols isis]
+   export agg_isis;
[edit]
+  policy-options {
+      policy-statement agg_isis {
+          term 1 {
+              from {
+                  protocol aggregate;
+                  route-filter 192.168.4.0/22 orlonger;
+              }
+              then accept;
+          }
+      }
+  }
```

And reachability to the new access network subnets is confirmed at R1:

```
{master}[edit]
jnpr@R1-RE0# run show route 192.168.7.0

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

192.168.4.0/22     *[IS-IS/165] 00:28:16, metric 30
> to 10.8.0.1 via xe-2/0/0.1

{master}[edit]
jnpr@R1-RE0#
```

The figure also provides a table that outlines the IFD, IFL-Set, and IFL level shaping; CIR; and per priority shaping plans. Per priority shaping for high-priority traffic is implemented at the IFD level to cap usage for the entire access network; priority-based shaping is also performed on some IFL-Sets to help enforce service differentiation between the tiers. Figure 5-35 provides a configuration-friendly view of the same H-CoS design.

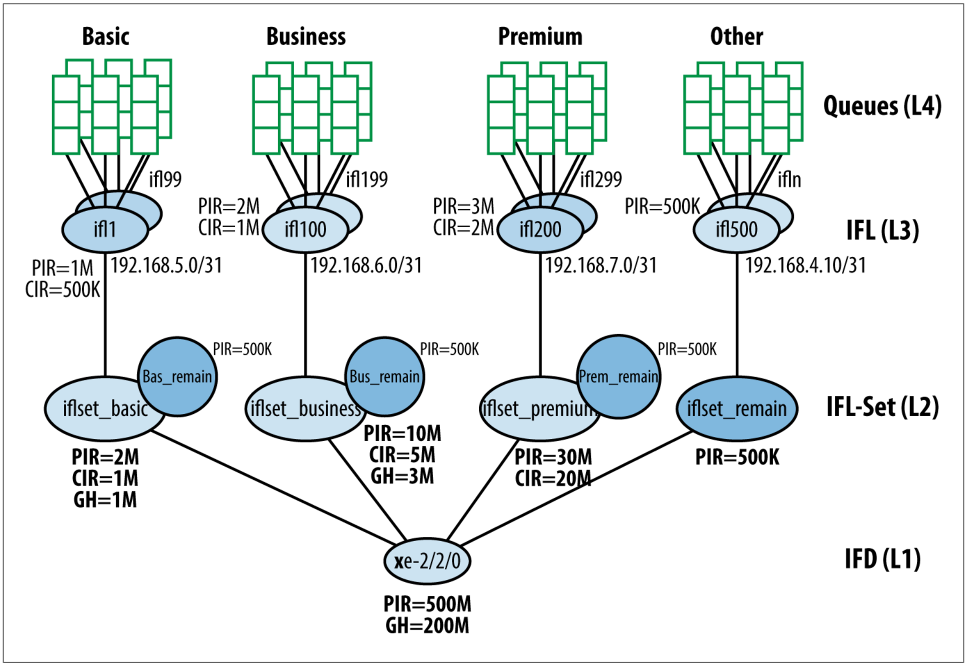


Figure 5-35. The H-CoS Configuration Plan.

Configure H-CoS

The actual configuration, at this stage of the chapter, is rather straightforward. The IFL definitions at R4 are shown. To save space, a single IFL is defined for each of the three sets, plus one extra to demonstrate use of a remaining profile. IFL 0 is left in place to provide a numbered link into the access network's DSLAM.

```
[edit]
jnpr@R4# show interfaces xe-2/2/0
hierarchical-scheduler;
vlan-tagging;
unit 0 {
    vlan-id 2000;
    family inet {
        address 192.168.4.2/30;
    }
}
unit 1 {
    vlan-id 1;
    family inet {
        address 192.168.5.0/31;
    }
}
unit 100 {
    vlan-id 100;
```

```
        family inet {
            address 192.168.6.0/31;
        }
    }
    unit 200 {
        vlan-id 200;
        family inet {
            address 192.168.7.0/31;
        }
    }
    unit 500 {
        vlan-id 500;
        family inet {
            address 192.168.4.10/31;
        }
    }
```

Note the presence of the `hierarchical-scheduler` command at the IFD level. Yeah baby! The interface sets are displayed next:

```
[edit]
jnpr@R4# show interfaces
interface-set iflset_basic {
    interface xe-2/2/0 {
        unit 1;
    }
}
interface-set iflset_business {
    interface xe-2/2/0 {
        unit 100;
    }
}
interface-set iflset_premium {
    interface xe-2/2/0 {
        unit 200;
    }
}
. . .
```

Not too much to see there. As the service grows, you keep adding IFLs from the designated ranges. If desired, you can use the `vlan-tags-outer` keyword to specify the outer tag, which for a dual tagged environment is an S-VLAN; otherwise, it's a C-VLAN. At a minimum, a set must have one member, and you cannot mix IFL and VLAN tag formats in the same set. Currently, ranges are not supported, but this is a likely upcoming enhancement. For now, you will have to enter all units/VLAN values individually, one at a time.

Next, the TCPs are displayed:

```
[edit]
jnpr@R4# show class-of-service traffic-control-profiles
tc-ifd-500m {
    shaping-rate 500m;
    overhead-accounting bytes -20;
    shaping-rate-priority-high 200m;
}
tc-iflset_basic {
    shaping-rate 2m;
    overhead-accounting bytes -20;
    shaping-rate-priority-high 1m;
    guaranteed-rate 1m;
}
tc-iflset_business {
    shaping-rate 10m;
    overhead-accounting bytes -20;
    shaping-rate-priority-high 3m;
    guaranteed-rate 5m;
}
tc-iflset_premium {
    shaping-rate 30m;
    overhead-accounting bytes -20;
    guaranteed-rate 20m;
}
tc-iflset_remain {
    scheduler-map sched_map_core;
    shaping-rate 500k;
    overhead-accounting bytes -20;
    shaping-rate-priority-high 500k;
}
tc-ifl_basic {
    scheduler-map sched_map_core;
    shaping-rate 1m;
    overhead-accounting bytes -20;
    guaranteed-rate 500k;
}
tc-ifl_business {
    scheduler-map sched_map_core;
    shaping-rate 2m;
    overhead-accounting bytes -20;
    guaranteed-rate 1m;
}
tc-ifl_premium {
    scheduler-map sched_map_core;
    shaping-rate 3m;
    overhead-accounting bytes -20;
    guaranteed-rate 2m;
}
tc-iflset_basic_remain {
    scheduler-map sched_map_core;
    shaping-rate 500k;
}
tc-iflset_business_remain {
    scheduler-map sched_map_core;
    shaping-rate 500k;
}
tc-iflset_premium_remain {
    scheduler-map sched_map_core;
    shaping-rate 500k;
}
```

Note that each level of scheduling, IFD, IFL-Set, and IFL is represented here. In addition, the IFD and the three interface sets each have a remaining profile configured. The TCPs that attach to IFLs include the scheduler map statement to provide the eight forwarding classes and scheduling parameters described previously. This example uses the same scheduler map, and therefore the same set of eight schedulers, for all IFLs. This is not a requirement. For example, you may want only two FCs (BE and NC) for the remaining profiles, or maybe you want different scheduler parameters for different users, even though those users might be in the same service tier. For example, a particular application may demand some specific buffer setting or might be very intolerant to loss, forcing a modified scheduler and a new scheduler map for that user.

Next is the class-of-service-level interface configuration. First, the IFL set definition, here used to tie a TCP to each set that was defined previously under the `[edit interfaces]` hierarchy:

```
[edit]
jnpr@R4# show class-of-service interfaces
interface-set iflset_basic {
    output-traffic-control-profile tc-iflset_basic;
    output-traffic-control-profile-remaining tc-iflset_basic_remain;
}
interface-set iflset_business {
    output-traffic-control-profile tc-iflset_business;
    output-traffic-control-profile-remaining tc-iflset_business_remain;
}
interface-set iflset_premium {
    output-traffic-control-profile tc-iflset_premium;
    output-traffic-control-profile-remaining tc-iflset_premium_remain;
}
```

And now, interface xe-2/2/0's CoS settings:

```
. . .
xe-2/2/0 {
    output-traffic-control-profile tc-ifd-500m;
    output-traffic-control-profile-remaining tc-iflset_remain;
    unit * {
        classifiers {
            dscp dscp_diffserv;
        }
        rewrite-rules {
            dscp dscp_diffserv;
        }
    }
    unit 0 {
        output-traffic-control-profile tc-ifl_business;
        classifiers {
            dscp dscp_diffserv;
        }
        rewrite-rules {
            dscp dscp_diffserv;
        }
    }
```

```
        unit 1 {
            output-traffic-control-profile tc-ifl_basic;
            classifiers {
                dscp dscp_diffserv;
            }
        }
        unit 100 {
            output-traffic-control-profile tc-ifl_business;
            classifiers {
                dscp dscp_diffserv;
            }
            rewrite-rules {
                dscp dscp_diffserv;
            }
        }
        unit 200 {
            output-traffic-control-profile tc-ifl_premium;
            classifiers {
                dscp dscp_diffserv;
            }
            rewrite-rules {
                dscp dscp_diffserv;
            }
        }
    }
```

Of note here is the specification of the IFD-level shaping and remaining TCPs. Also, note that unit 500 is *not* mentioned explicitly; this is a key point, as it means unit 500 is not specified in any interface set, nor does it have a TCP applied under `[edit class-of-service interfaces]` hierarchy. As a result, we expect unit 500 to be serviced by the IFD-level remaining profile. The use of a wild-card unit allows application of the DSCP rewrite and classifiers to an interface that is not explicitly listed, such as IFL 500. Note how units 1, 100, and 200 each have an IFL-level TCP applied, and that they are all listed in an IFL-Set. The result in these IFLs will become level 3, and they will attach to their associated interface set at level 2.

Verify H-CoS

You verify H-CoS in the same manner as shown previously in the per unit scheduling example. The only real difference is the L2 scheduling nodes and presence of interface sets. We begin with operational mode verification. First, the interface sets are confirmed:

```
jnpr@R4# run show class-of-service interface-set
Interface-set: iflset_basic, Index: 4
Physical interface: xe-2/2/0, Index: 152
Queues supported: 8, Queues in use: 8
  Output traffic control profile: tc-iflset_basic, Index: 61878

Interface-set: iflset_business, Index: 5
Physical interface: xe-2/2/0, Index: 152
Queues supported: 8, Queues in use: 8
```

```
    Output traffic control profile: tc-iflset_business, Index: 25290

  Interface-set: iflset_premium, Index: 6
  Physical interface: xe-2/2/0, Index: 152
  Queues supported: 8, Queues in use: 8
    Output traffic control profile: tc-iflset_premium, Index: 23149
```

And now, the various TCPs:

```
[edit]
jnpr@R4# run show class-of-service traffic-control-profile
Traffic control profile: tc-ifd-500m, Index: 17194
  Shaping rate: 500000000
  Shaping rate priority high: 200000000
  Scheduler map: <default>

Traffic control profile: tc-ifl_basic, Index: 2288
  Shaping rate: 1000000
  Scheduler map: sched_map_core
  Guaranteed rate: 500000

Traffic control profile: tc-ifl_business, Index: 7785
  Shaping rate: 2000000
  Scheduler map: sched_map_core
  Guaranteed rate: 1000000

Traffic control profile: tc-ifl_premium, Index: 16776
  Shaping rate: 3000000
  Scheduler map: sched_map_core
  Guaranteed rate: 2000000

Traffic control profile: tc-iflset_basic, Index: 61878
  Shaping rate: 2000000
  Shaping rate priority high: 1000000
  Scheduler map: <default>
  Guaranteed rate: 1000000

Traffic control profile: tc-iflset_basic_remain, Index: 42633
  Shaping rate: 500000
  Scheduler map: sched_map_core

Traffic control profile: tc-iflset_business, Index: 25290
  Shaping rate: 10000000
  Shaping rate priority high: 3000000
  Scheduler map: <default>
  Guaranteed rate: 5000000

Traffic control profile: tc-iflset_business_remain, Index: 15725
  Shaping rate: 500000
  Scheduler map: sched_map_core

Traffic control profile: tc-iflset_premium, Index: 23149
  Shaping rate: 30000000
  Scheduler map: <default>
  Guaranteed rate: 20000000
```

```
Traffic control profile: tc-iflset_premium_remain, Index: 63572
  Shaping rate: 500000
  Scheduler map: sched_map_core

Traffic control profile: tc-iflset_remain, Index: 14271
  Shaping rate: 500000
  Shaping rate priority high: 500000
  Scheduler map: sched_map_core
```

Note how the TCPs that are used for L2 or L1 scheduling nodes omit the `scheduler-map` statements, and so are shown using the default mapping. The `tc-iflset_remain` TCP is applied to the IFD, but it functions as a type of level 2 interface set, so using the `scheduler-map` statement here makes eight queues available for sharing among all IFLs that fall into the remaining group. The CoS settings for the xe-2/2/0 interface are displayed:

```
[edit]
jnpr@R4# run show class-of-service interface xe-2/2/0
Physical interface: xe-2/2/0, Index: 152
Queues supported: 8, Queues in use: 8
Total non-default queues created: 56
  Output traffic control profile: tc-ifd-500m, Index: 17194
  Congestion-notification: Disabled

  Logical interface: xe-2/2/0.0, Index: 327, Dedicated Queues: yes
    Object                  Name                   Type                    Index
    Traffic-control-profile tc-ifl_business        Output                   7785
    Rewrite                 dscp_diffserv          dscp                    23080
    Classifier              dscp_diffserv          dscp                    23080

  Logical interface: xe-2/2/0.1, Index: 328, Dedicated Queues: yes
    Object                  Name                   Type                    Index
    Traffic-control-profile tc-ifl_basic           Output                   2288
    Classifier              dscp_diffserv          dscp                    23080

  Logical interface: xe-2/2/0.100, Index: 329, Dedicated Queues: yes
    Object                  Name                   Type                    Index
    Traffic-control-profile tc-ifl_business        Output                   7785
    Rewrite                 dscp_diffserv          dscp                    23080
    Classifier              dscp_diffserv          dscp                    23080

  Logical interface: xe-2/2/0.200, Index: 330, Dedicated Queues: yes
    Object                  Name                   Type                    Index
    Traffic-control-profile tc-ifl_premium         Output                  16776
    Rewrite                 dscp_diffserv          dscp                    23080
    Classifier              dscp_diffserv          dscp                    23080

  Logical interface: xe-2/2/0.32767, Index: 332

  Logical interface: xe-2/2/0.500, Index: 331
    Object                  Name                   Type                    Index
    Rewrite                 dscp_diffserv          dscp                    23080
    Classifier              dscp_diffserv          dscp                    23080
```

Of note here is how unit 500 is not shown as having any TCP/scheduler applied. All other units of this interface are specified as having a TCP, which in turn has the scheduler map to provide queues to the IFL. The key point, again, is that unit 500 is not listed in any interface set, so will not be subjected to any set level remaining profile. This unit does not have its own TCP applied, so the only way it will get its queues, and the bandwidth they afford, is to use the interface-level remaining profile.

The real action is in the PFE. The resulting H-CoS scheduler hierarchy is displayed:

```
NPC2(R4 vty)# sho cos scheduler-hierarchy

class-of-service EGRESS scheduler hierarchy - rates in kbps
--------------------------------------------------------------------------------
                                   shaping guarntd delaybf  excess
interface name               index    rate    rate    rate    rate      other
---------------------------- ----- ------- ------- ------- ------- -------------
xe-2/0/0                       148       0       0       0       0
xe-2/0/1                       149       0       0       0       0
xe-2/1/0                       150       0       0       0       0
xe-2/1/1                       151       0       0       0       0
xe-2/2/0                       152  500000       0       0       0
  iflset_basic                   4    2000    1000       0       0
    xe-2/2/0.1                 328    1000     500       0       0
      q 0 - pri 0/0          20205       0      5%       0     35%
      q 1 - pri 0/0          20205       0      5%       0      5%
      q 2 - pri 0/0          20205       0     10%       0     10%
      q 3 - pri 3/1          20205       0      5%     10%      5%
      q 4 - pri 0/0          20205       0     30%       0     30%
      q 5 - pri 4/0          20205       0     30%   25000      0%      exact
      q 6 - pri 0/0          20205       0     15%       0     15%
      q 7 - pri 2/5          20205       0       0       0      0%
    iflset_basic-rtp             4     500       0       0       0
      q 0 - pri 0/0          20205       0      5%       0     35%
      q 1 - pri 0/0          20205       0      5%       0      5%
      q 2 - pri 0/0          20205       0     10%       0     10%
      q 3 - pri 3/1          20205       0      5%     10%      5%
      q 4 - pri 0/0          20205       0     30%       0     30%
      q 5 - pri 4/0          20205       0     30%   25000      0%      exact
      q 6 - pri 0/0          20205       0     15%       0     15%
      q 7 - pri 2/5          20205       0       0       0      0%
  iflset_business                5   10000    5000       0       0
    xe-2/2/0.100               329    2000    1000       0       0
      q 0 - pri 0/0          20205       0      5%       0     35%
      q 1 - pri 0/0          20205       0      5%       0      5%
      q 2 - pri 0/0          20205       0     10%       0     10%
      q 3 - pri 3/1          20205       0      5%     10%      5%
      q 4 - pri 0/0          20205       0     30%       0     30%
      q 5 - pri 4/0          20205       0     30%   25000      0%      exact
      q 6 - pri 0/0          20205       0     15%       0     15%
      q 7 - pri 2/5          20205       0       0       0      0%
    iflset_business-rtp          5     500       0       0       0
      q 0 - pri 0/0          20205       0      5%       0     35%
      q 1 - pri 0/0          20205       0      5%       0      5%
      q 2 - pri 0/0          20205       0     10%       0     10%
```

```
          q 3 - pri 3/1           20205        0      5%     10%      5%
          q 4 - pri 0/0           20205        0     30%       0     30%
          q 5 - pri 4/0           20205        0     30%   25000      0%     exact
          q 6 - pri 0/0           20205        0     15%       0     15%
          q 7 - pri 2/5           20205        0       0       0      0%
      iflset_premium                  6    30000   20000       0       0
        xe-2/2/0.200                330     3000    2000       0       0
          q 0 - pri 0/0           20205        0      5%       0     35%
          q 1 - pri 0/0           20205        0      5%       0      5%
          q 2 - pri 0/0           20205        0     10%       0     10%
          q 3 - pri 3/1           20205        0      5%     10%      5%
          q 4 - pri 0/0           20205        0     30%       0     30%
          q 5 - pri 4/0           20205        0     30%   25000      0%     exact
          q 6 - pri 0/0           20205        0     15%       0     15%
          q 7 - pri 2/5           20205        0       0       0      0%
        iflset_premium-rtp            6      500       0       0       0
          q 0 - pri 0/0           20205        0      5%       0     35%
          q 1 - pri 0/0           20205        0      5%       0      5%
          q 2 - pri 0/0           20205        0     10%       0     10%
          q 3 - pri 3/1           20205        0      5%     10%      5%
          q 4 - pri 0/0           20205        0     30%       0     30%
          q 5 - pri 4/0           20205        0     30%   25000      0%     exact
          q 6 - pri 0/0           20205        0     15%       0     15%
          q 7 - pri 2/5           20205        0       0       0      0%
      xe-2/2/0.0                    327     2000    1000       0       0
        q 0 - pri 0/0             20205        0      5%       0     35%
        q 1 - pri 0/0             20205        0      5%       0      5%
        q 2 - pri 0/0             20205        0     10%       0     10%
        q 3 - pri 3/1             20205        0      5%     10%      5%
        q 4 - pri 0/0             20205        0     30%       0     30%
        q 5 - pri 4/0             20205        0     30%   25000      0%     exact
        q 6 - pri 0/0             20205        0     15%       0     15%
        q 7 - pri 2/5             20205        0       0       0      0%
      xe-2/2/0-rtp                  152      500       0       0       0
        q 0 - pri 0/0             20205        0      5%       0     35%
        q 1 - pri 0/0             20205        0      5%       0      5%
        q 2 - pri 0/0             20205        0     10%       0     10%
        q 3 - pri 3/1             20205        0      5%     10%      5%
        q 4 - pri 0/0             20205        0     30%       0     30%
        q 5 - pri 4/0             20205        0     30%   25000      0%     exact
        q 6 - pri 0/0             20205        0     15%       0     15%
        q 7 - pri 2/5             20205        0       0       0      0%
```

There is a lot of information here, but the only thing that is new in relation to previous displays is the presence of interface sets. The **iflset_basic** set is given index 4, the related IFL, unit 1, is indexed with 328, while the IFD itself is 152; these values are used later to poke a little deeper, so keep them in mind. Note how each IFL-Set has a remaining traffic profile (RTP) that offers eight shared queues. Note how all IFL and IFL-Sets display the configured PIR and CIR information. The basic set on IFL 1, for example, shows 1 Mbps of PIR and 500 kbps of CIR.

All queues use the same scheduling policy index, given the same scheduler map is used for all IFLs. Again, this is not a requirement, but again, CoS is all about consistency, so there is something to be said for the common scheduler map model being demonstrated here as well. The IFD-level remaining profile is index 152. Note the 500 kbps PIR, again matching the deployment plans.

The CoS settings for the IFD are displayed:

```
NPC2(R4 vty)# sho cos ifd-entry  152
CoS IFD IDX: 152
Port Speed: 10000000000
Scheduler Mode: COS_IFD_SCHED_HIER_SCHED_MODE
IF toolkit scheduler mode: ifd_has_hier_sched:TRUE ifd_has_2level_hier_sched:FALSE
scheduler_map_id[       egress] : 20205
EGRESS Traffic Params
----------------------------------------------
      Speed              : 500000000
      Total bw           : 135500000
      bw_remain          : 364500000
      g_bw_remain        : 469500000
      delay_bw_remain    : 500000000
      oversubscribed     : FALSE
      num_ifl_default_bw : 0
      num_ifl_gbw        : 7 (PIR/CIR)
      num_ifl_ebw        : 0
      max_shaping_rate   : 60000000
      max_g_rate         : 40000000
Shaping      Guaranteed   Delay-Buffer  Excess
rate         rate         rate          rate
-----------  -----------  ------------  -------
  500000000    500000000     500000000       0

EGRESS Remaining Traffic Params
----------------------------------------------
Shaping      Guaranteed   Delay-Buffer  Excess
rate         rate         rate          rate
-----------  -----------  ------------  -------
     500000            0             0       0
```

Of note is the section on egress remaining traffic parameters, which confirms correct attachment of the remaining profile. The correct speed of 500 Mbps and the interface's PIR/CIR mode is confirmed here as well. Per priority shaping at the IFD is displayed:

```
NPC2(R4 vty)# sho cos ifd-per-priority-shaping-rates

EGRESS IFD Per-priority shaping rates, in kbps
                                     per-priority shaping-rates (in kbps)
       ----------------------------- ---------------------------------------
Ifd     Shaping   Guarantd  DelayBuf    GH      GM      GL      EH      EL
Index     Rate      Rate      Rate     Rate    Rate    Rate    Rate    Rate
------ --------- --------- --------- ------- ------- ------- ------- -------
   148         0  10000000  10000000       0       0       0       0       0
   149         0  10000000  10000000       0       0       0       0       0
   150         0  10000000  10000000       0       0       0       0       0
   151         0  10000000  10000000       0       0       0       0       0
   152    500000    500000    500000  200000       0       0       0       0
   153         0  10000000  10000000       0       0       0       0       0
```

```
EGRESS IFD Remaining-traffic shaping rates, in kbps
                                     per-priority shaping-rates (in kbps)
       ----------------------------- ---------------------------------------
Ifd     Shaping   Guarantd  DelayBuf   GH      GM      GL      EH      EL
Index    Rate      Rate      Rate     Rate    Rate    Rate    Rate    Rate
------ --------- --------- --------- ------- ------- ------- ------- -------
   152       500         0         0     500       0       0       0       0
```

This display confirms both the PIR and the high-priority shaping rates. Next, information on an interface set, which is at level 2 of the H-CoS hierarchy, is displayed:

```
NPC2(R4 vty)# sho cos iflset-entry 4
EGRESS Traffic Params for interface-set index 4
-------------------------------------------------------
Parent ifd:                xe-2/2/0
Shaping-rate:                  2000 kbps
Shaping-rate-burst:          <none>
Guaranteed-rate:               1000 kbps
Guaranteed-rate-burst:       <none>
Delay-buffer-rate:           <none>
Excess-rate:                 <none>
Excess-rate-high:            <none>
Excess-rate-low:             <none>
Shaping-rate-pri-GH:           1000 kbps
Shaping-rate-pri-GM:         <none>
Shaping-rate-pri-GL:         <none>
Shaping-rate-pri-EH:         <none>
Shaping-rate-pri-EL:         <none>
Adjust-min-shaping-rate:     <none>
Adjust-delta-sr:             <none>
Scale-factor:                <none>

EGRESS Remaining Traffic Params for interface-set index 4
-------------------------------------------------------
RT Shaping-rate:                500 kbps
RT Shaping-rate-burst:       <none>
RT Guaranteed-rate:          <none>
RT Guaranteed-rate-burst:    <none>
RT Delay-buffer-rate:        <none>
RT Excess-rate:              <none>
RT Excess-rate-high:         <none>
RT Excess-rate-low:          <none>
```

This output references index 4, which is the level 2 node assigned to the `ifl set_basic` interface set. Once again, the expected PIR, CIR, and priority-based shaping is in effect. The set's remaining profile is also confirmed at a 500 kbps PIR. Having seen the level 1 IFD and the level 2 IFL-Set, it seems like it's time to look at an IFL belonging to the `iflset_basic` set to complete the tour of scheduling levels:

```
NPC2(R4 vty)# sho cos ifl-entry 328
CoS IFL IDX: 328
  CoS IFLSET IDX: 4
    CoS IFD IDX: 152
Flags: 0x0
```

```
CoS Flags: 0x0
classifier[         DSCP] : 23080
scheduler_map_id[       egress] : 20205
EGRESS Traffic Params
-------------------------------------------------------

Shaping      Guaranteed   Delay-Buffer  Excess   Excess
rate         rate         rate          rate-hi  rate-lo
-----------  -----------  ------------  -------  -------
    1000000       500000        500000       12       12
```

This display is for IFL 1, which currently resides in the `iflset_basic` set. Note the IFL level's PIR and CIR rates match the values shown in the figure. While we are here, the other IFL-level TCPs are displayed to provide contrast:

```
NPC2(R4 vty)# sho cos ifl-tc-profile
INGRESS Traffic Params
Ifl    Shaping     Guaranteed  Delay-Buffer Excess  Excess  Ovrhd Ovrhd Adjust
index  rate        rate        rate         rate-hi rate-lo mode  bytes min
------ ----------- ----------- ------------ ------- ------- ----- ----- ------
EGRESS Traffic Params
Ifl    Shaping     Guaranteed  Delay-Buffer Excess  Excess  Ovrhd Ovrhd Adjust
index  rate        rate        rate         rate-hi rate-lo mode  bytes min
------ ----------- ----------- ------------ ------- ------- ----- ----- ------
   327     2000000     1000000      1000000      25      25 Frame   -20      0
   328     1000000      500000       500000      12      12 Frame   -20      0
   329     2000000     1000000      1000000      25      25 Frame   -20      0
   330     3000000     2000000      2000000      50      50 Frame   -20
```

The output makes it clear that IFLs in the different service tiers should receive varying levels of PIR and CIR. Indexes 328 to 330 represent IFLs 1, 100, and 200. Note that the premium class IFL with index 330 has three times the guaranteed rate of a basic user. The display also confirms the 20-byte adjustment made to overhead accounting.

The `halp` switch is added to get the hardware-specific view, and IFL-Set information is displayed. Note how a L1 and L2 scheduler index is provided for each L3 IFL-Set:

```
NPC2(R4 vty)# sho cos halp iflset all
================================================================================
Interface Count: 3
================================================================================

--------------------------------------------------------------------------------
IFLSET name: (iflset_basic, xe-2/2/0)   (Index 4, IFD Index 152)
    QX chip id: 1
    QX chip L2 index: 2
    QX chip L3 index: 1
    QX chip base Q index: 8
Queue    State        Max       Guaranteed   Burst  Weight Priorities Drop-Rules
Index                 rate         rate      size            G    E   Wred  Tail
------ ----------- ----------- ------------ ------- ------ ---------- ----------
     8  Configured      500000            0    8192    350   GL   EL     5     0
     9  Configured      500000            0    8192     50   GL   EL     6     0
    10  Configured      500000            0    8192    100   GL   EL     7     0
    11  Configured      500000            0    8192     50   GH   EL     4   191
    12  Configured      500000                 0    8192    300    GL   EL    8     0
    13  Configured      500000          Disabled    8192      1    GH   EH    4   191
    14  Configured      500000                 0    8192    150    GL   EL    9     0
    15  Configured      500000                 0    8192      1    GM   EN    4   127

Rate limit info:
    Q 5: Bandwidth = 150000, Burst size = 15000. Policer NH: 0x3064172200140000

    Index NH: 0xda4be05537001006

--------------------------------------------------------------------------------
IFLSET name: (iflset_business, xe-2/2/0)   (Index 5, IFD Index 152)
    QX chip id: 1
    QX chip L2 index: 3
    QX chip L3 index: 2
    QX chip base Q index: 16
Queue    State        Max       Guaranteed   Burst  Weight Priorities Drop-Rules
Index                 rate         rate      size            G    E   Wred  Tail
------ ----------- ----------- ------------ ------- ------ ---------- ----------
    16  Configured      500000                 0    8192    350    GL   EL    5     0
    17  Configured      500000                 0    8192     50    GL   EL    6     0
    18  Configured      500000                 0    8192    100    GL   EL    7     0
    19  Configured      500000                 0    8192     50    GH   EL    4   191
    20  Configured      500000                 0    8192    300    GL   EL    8     0
    21  Configured      500000          Disabled    8192      1    GH   EH    4   191
    22  Configured      500000                 0    8192    150    GL   EL    9     0
    23  Configured      500000                 0    8192      1    GM   EN    4   127

Rate limit info:
    Q 5: Bandwidth = 150000, Burst size = 15000. Policer NH: 0x3064171a00141000

    Index NH: 0xda4be05522001006

--------------------------------------------------------------------------------
IFLSET name: (iflset_premium, xe-2/2/0)   (Index 6, IFD Index 152)
    QX chip id: 1
    QX chip L2 index: 4
    QX chip L3 index: 3
    QX chip base Q index: 24
Queue    State        Max       Guaranteed   Burst  Weight Priorities Drop-Rules
Index                 rate         rate      size            G    E   Wred  Tail
------ ----------- ----------- ------------ ------- ------ ---------- ----------
    24  Configured      500000                 0    8192    350    GL   EL    5     0
    25  Configured      500000                 0    8192     50    GL   EL    6     0
    26  Configured      500000                 0    8192    100    GL   EL    7     0
    27  Configured      500000                 0    8192     50    GH   EL    4   191
    28  Configured      500000                 0    8192    300    GL   EL    8     0
    29  Configured      500000          Disabled    8192      1    GH   EH    4   191
    30  Configured      500000                 0    8192    150    GL   EL    9     0
    31  Configured      500000                 0    8192      1    GM   EN    4   127

Rate limit info:
    Q 5: Bandwidth = 150000, Burst size = 15000. Policer NH: 0x3064171200147000

    Index NH: 0xda4be0552c801006
```

And, in like fashion, an IFL-level display:

```
NPC2(R4 vty)# sho cos halp ifl all
================================================================================
Interface Count: 4
================================================================================

. . .

--------------------------------------------------------------------------------
IFL name: (xe-2/2/0.1, xe-2/2/0)   (Index 328, IFD Index 152)
    QX chip id: 1
    QX chip dummy L2 index: -1
    QX chip L3 index: 7
    QX chip base Q index: 56
Queue     State         Max       Guaranteed   Burst  Weight Priorities Drop-Rules
Index                   rate         rate      size              G    E   Wred  Tail
------ ----------- ----------- ------------ ------- ------ ---------- ----------
    56  Configured     1000000        25000   16384    350   GL   EL    5      1
    57  Configured     1000000        25000   16384     50   GL   EL    6      1
    58  Configured     1000000        50000   16384    100   GL   EL    7      1
    59  Configured     1000000        25000   16384     50   GH   EL    4    191
    60  Configured     1000000       150000   16384    300   GL   EL    8      1
    61  Configured     1000000     Disabled   16384      1   GH   EH    4    191
    62  Configured     1000000        75000   16384    150   GL   EL    9      1
    63  Configured     1000000            0   16384      1   GM   EN    4    128

Rate limit info:
    Q 5: Bandwidth = 300000, Burst size = 30000. Policer NH: 0x8a6d83800020000

    Index NH: 0xda4be05501001006

--------------------------------------------------------------------------------
IFL name: (xe-2/2/0.100, xe-2/2/0)   (Index 329, IFD Index 152)
    QX chip id: 1
    QX chip dummy L2 index: -1
    QX chip L3 index: 8
    QX chip base Q index: 64
Queue     State         Max       Guaranteed   Burst  Weight Priorities Drop-Rules
Index                   rate         rate      size              G    E   Wred  Tail
------ ----------- ----------- ------------ ------- ------ ---------- ----------
    64  Configured     2000000        50000   32768    350   GL   EL    5      2
    65  Configured     2000000        50000   32768     50   GL   EL    6      2
    66  Configured     2000000       100000   32768    100   GL   EL    7      2
    67  Configured     2000000        50000   32768     50   GH   EL    4    192
    68  Configured     2000000       300000   32768    300   GL   EL    8      2
    69  Configured     2000000     Disabled   32768      1   GH   EH    4    192
    70  Configured     2000000       150000   32768    150   GL   EL    9      2
    71  Configured     2000000            0   32768      1   GM   EN    4    129

Rate limit info:
    Q 5: Bandwidth = 600000, Burst size = 60000. Policer NH: 0x8a6d91000020000

    Index NH: 0xda4be0550a801006

--------------------------------------------------------------------------------
```

```
IFL name: (xe-2/2/0.200, xe-2/2/0)   (Index 330, IFD Index 152)
    QX chip id: 1
    QX chip dummy L2 index: -1
    QX chip L3 index: 9
    QX chip base Q index: 72
Queue    State         Max       Guaranteed   Burst  Weight Priorities Drop-Rules
Index                  rate         rate      size            G    E   Wred  Tail
------ ----------- ----------- ------------ ------- ------ ---------- ----------
    72  Configured    3000000       100000   65536    350   GL   EL     5     3
    73  Configured    3000000       100000   65536     50   GL   EL     6     3
    74  Configured    3000000       200000   65536    100   GL   EL     7     3
    75  Configured    3000000       100000   65536     50   GH   EL     4   193
    76  Configured    3000000       600000   65536    300   GL   EL     8     3
    77  Configured    3000000     Disabled   65536      1   GH   EH     4   193
    78  Configured    3000000       300000   65536    150   GL   EL     9     3
    79  Configured    3000000            0   65536      1   GM   EN     4   130

Rate limit info:
    Q 5: Bandwidth = 900000, Burst size = 90000. Policer NH: 0x8a6d9e800020000

    Index NH: 0xda4be05516801006
```

The IFL-level output again confirms the different service tiers parameters are in effect. Note how IFL 500 is missing from all these displays. This is in keeping with its lack of CoS configuration, and again is the reason we have the IFD-level remaining profile in place. Currently, H-CoS queuing is handled by the fine-grained queuing block, a function performed by the QX ASIC. For the sake of completeness, the L1 scheduling node at the IFD level is displayed. This node is shared by all IFL-Sets:

```
NPC2(R4 vty)# sho qxchip 1 l1 1
L1 node configuration   : 1
        state           : Configured
        child_l2_nodes  : 5
        config_cache    : 21052000
        rate_scale_id   : 0
        gh_rate         : 200000000, burst-exp 18 (262144 bytes scaled by 16)
        gm_rate         : 0
        gl_rate         : 0
        eh_rate         : 0
        el_rate         : 0
        max_rate        : 500000000
        cfg_burst_size  : 8388608 bytes
        burst_exp       : 19 (524288 bytes scaled by 16)
        byte_adjust     : 4
        cell_mode       : off
        pkt_adjust      : 0
```

And now a L2 node, in this case for the `iflset_business` IFL-Set:

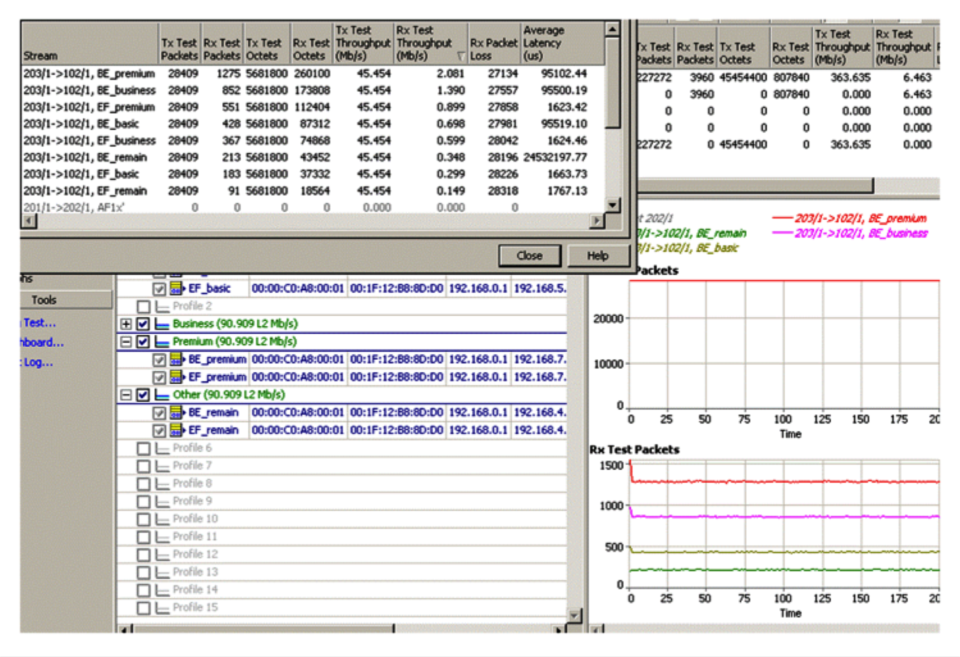
```
NPC2(R4 vty)# sho qxchip 1 l2 3
L2 node configuration   : 3
        state           : Configured
        child_l3_nodes  : 2
        l1_index        : 1
        config_cache[0] : 00000000
```

```
config_cache[1] : 000023e8
config_cache[2] : fb0fc100
rw_scale_id     : 0
gh_rate         : 3000000, burst-exp 12 (4096 bytes scaled by 16)
gm_rate         : 0
gl_rate         : 0
eh_rate         : 0
el_rate         : 0
max_rate        : 10000000
g_rate_enable   : TRUE
g_rate          : 5000000
cfg_burst_size  : 131072 bytes
burst_exp       : 13 (8192 bytes scaled by 16)
eh_weight       : 125
el_weight       : 125
byte_adjust     : 4
cell_mode       : off
gh_debit_to_g_rate : TRUE
gm_debit_to_g_rate : TRUE
eh_promo_to_g : TRUE
el_promo_to_g : TRUE
```

The output confirms the L2 node's G-Rate and PIR settings. Also, the per priority shaper and the default priority handling flags, here confirming that excess can be promoted into GL, and that GH/GM traffic is debited from the node's G-Rate, but not eligible for actual demotion, are also confirmed.

Verify H-CoS in the Data Plane

With the operational and shell level commands returning the expected values, it appears that H-CoS is up and running as per the design requirement and parameters shown in Figure 5-35. To actually measure data plane behavior, the router tester is modified to generate L3 traffic using four different profiles. Each profile consists of two streams, BE and EF; there are four such profiles so that traffic can be sent to a member of each service tier and to the remaining profile. The profiles generate both streams at a combined Layer 2 rate of 90.909 Mbps. Figure 5-36 shows the measured results.



Stream	Tx Test Packets	Rx Test Packets	Tx Test Octets	Rx Test Octets	Tx Test Throughput (Mb/s)	Rx Test Throughput (Mb/s)	Rx Packet Loss	Average Latency (us)
203/1->102/1, BE_premium	28409	1275	5681800	260100	45.454	2.081	27134	95102.44
203/1->102/1, BE_business	28409	852	5681800	173808	45.454	1.390	27557	95500.19
203/1->102/1, EF_premium	28409	551	5681800	112404	45.454	0.899	27858	1623.42
203/1->102/1, BE_basic	28409	428	5681800	87312	45.454	0.698	27981	95519.10
203/1->102/1, EF_business	28409	367	5681800	74868	45.454	0.599	28042	1624.46
203/1->102/1, BE_remain	28409	213	5681800	43452	45.454	0.348	28196	24532197.77
203/1->102/1, EF_basic	28409	183	5681800	37332	45.454	0.299	28226	1663.73
203/1->102/1, EF_remain	28409	91	5681800	18564	45.454	0.149	28318	1767.13
201/1->202/1, AF1x'	0	0	0	0	0.000	0.000	0	

x Test Packets	Rx Test Packets	Tx Test Octets	Rx Test Octets	Tx Test Throughput (Mb/s)	Rx Test Throughput (Mb/s)
227272	3960	45454400	807840	363.635	6.463
0	3960	0	807840	0.000	6.463
0	0	0	0	0.000	0.000
0	0	0	0	0.000	0.000
227272	0	45454400	0	363.635	0.000

☑ EF_basic	00:00:C0:A8:00:01	00:1F:12:B8:8D:D0	192.168.0.1	192.168.5.
□ Profile 2				
⊞ ☑ Business (90.909 L2 Mb/s)				
⊟ ☑ Premium (90.909 L2 Mb/s)				
☑ BE_premium	00:00:C0:A8:00:01	00:1F:12:B8:8D:D0	192.168.0.1	192.168.7.
☑ EF_premium	00:00:C0:A8:00:01	00:1F:12:B8:8D:D0	192.168.0.1	192.168.7.
⊟ ☑ Other (90.909 L2 Mb/s)				
☑ BE_remain	00:00:C0:A8:00:01	00:1F:12:B8:8D:D0	192.168.0.1	192.168.4.
☑ EF_remain	00:00:C0:A8:00:01	00:1F:12:B8:8D:D0	192.168.0.1	192.168.4.
□ Profile 6				
□ Profile 7				
□ Profile 8				
□ Profile 9				
□ Profile 10				
□ Profile 11				
□ Profile 12				
□ Profile 13				
□ Profile 14				
□ Profile 15				

Figure 5-36. H-CoS Measured Results.

The lower right portion of the display shows a line graph of BE through for all four profiles. From bottom (slowest) to the top, we have BE remaining, BE basic, BE business, and BE premium. The throughput rates shown in the upper left make it clear that a business subscriber gets two times that of a basic user and that a premium user gets three times that amount, receiving 1,500 PPS.

The upper right indicates the transmitted and received traffic rates. The receive throughput value of 6.4 Mbps is quite interesting, given that we have maximum rate traffic flowing on four IFLs; one for basic, (1 Mbps), one for business (2 Mbps), one for premium (3 Mbps), and the final IFL for the remaining user (500 kbps). The aggregate rate of 6.4 Mbps confirms that all traffic profiles are correctly configured, and H-CoS is working properly in the test network.

The tabular data in the upper left confirms loss for all IFLs/streams, which is expected given the input rate of 90 Mbps and the IFLs shaped to an aggregate of 3 Mbps with a 0.5 Mbps reserve for remaining. Recall that EF is set to 30% and rate limited. As such, the 30% is based on IFL shaping, not committed rate. Note that the EF stream for the basic user represents approximately 30% of 1 Mbps at 0.3 Mbps. In contrast, the business user gets two times that, or 0.6 Mbps. And as expected, the premium user gets the most real-time traffic at about 0.9 Mbps. Summing the EF and BE traffic shows that each IFL is receiving its full-shaped bandwidth and clearly represents the service differences between basic and premium.

This concludes the operational verification of Junos H-CoS.

Trio CoS Summary

When you combine Junos features with Trio-based PFEs, the world becomes your oyster. This chapter covered Trio CoS capabilities and current scaling capabilities, with focus on Trio scheduling and leftover bandwidth sharing. Port mode and per unit scheduling were covered, but heavy emphasis was placed on the operation and design characteristics of hierarchical CoS.

Having arrived here, you should be able to analyze an H-CoS configuration and predict queue- and IFL-level throughput for both PIR and CIR/PIR mode interfaces. This is no easy feat, and a self-high-five is in order for having stuck it out to the end of this rather long chapter. If only there was a LLQ way to convey all this H-CoS, a great many trees could have been saved.

Chapter Review Questions

1. What MPC types support per unit scheduling mode?
    a. MPC1
    b. MPC1 Q
    c. MPC2E
    d. MPC3E
2. Which option controls excess bandwidth sharing in Trio?
    a. Excess rate
    b. Excess bandwidth sharing
    c. This is not under user control, always shared in proportion to transmit weight
    d. This is not under user control, always shared in proportion to shaping weight
3. When shaping or displaying interface queue statistics, what overhead does Trio take into account?
    a. Layer 2 only
    b. Layer 1 and Layer 2
    c. Layer 3 only
    d. Layer 2 and Layer 3
    e. None of the above
4. Which of the below configures a guaranteed rate?
    a. Transmit rate for a queue on a nonoversubscribed IFL
    b. Adding peak information rate to a Traffic Control Profile applied to an IFL-Set

   c. Adding committed information rate to a Traffic Control Profile applied to an IFL-Set
   d. Adding committed information rate to a Traffic Control Profile applied to an IFD
   e. Both A and C
5. Which of the below are true with regards to Trio H-CoS?
   a. You can overbook guaranteed rates
   b. The sum of queue transmit rates can exceed IFL shaping speed
   c. The shaping rate of level 3 scheduler can be higher than the underlying shaped rate of the level 1 or level 2 node
   d. Remaining profiles can be used to provide CoS to IFLs either at the IFL-Set of IFD levels
   e. All of the above
6. Which is true regarding priority inheritance?
   a. Strict-high queues are demoted once they reach their shaped or policed rate
   b. A GL/low-priority queue can never be demoted
   c. A queue can demote GH, GM, or GL, but scheduler nodes can only demote GL in regards to guaranteed rates
   d. A scheduler node can demote any priority level as a function of per priority shaping
   e. Both C and D
7. Which of the following are supported for AE interfaces in 11.4?
   a. Equal share mode is supported
   b. Replicated mode is supported
   c. Interface sets (IFL-Sets)
   d. You cannot use H-CoS over AE interfaces
   e. Both A and B
8. Which of the following can be used to cap a queues bandwidth usage below the IFL shaping rate?
   a. Exact
   b. Rate limit
   c. Excess-priority none, combined with high priority
   d. Shaping rate
   e. All of the above
9. Which is true regarding scheduler maps and Traffic Control Profiles?
   a. You use a TCP to set shaping rates for a queue

b. Scheduler maps link one or more schedulers to a level 1 or level 2 node

c. A TCP is a CoS container that can reference a scheduler map to support queues

d. A TCP does not use a scheduler map when applied to a scheduling node

e. Both C and D

10. Which is true regarding the `delay-buffer-rate` parameter in a TCP?

a. The delay buffer rate allows you to override default of a delay buffer that's based on G-Rate or shaping rate when G-Rate is not set

b. A larger delay buffer means less loss but more delay when dealing with bursty traffic

c. The queue level buffer size statement is used to assign a queue some portion of the delay buffer used by the underlying IFL

d. Trio uses a dynamic delay buffer that allows borrowing from other queues that do not need their allocated buffer

e. A, B, and C

f. B and C

11. Which is true based on the output provided?

```
PC2(R4 vty)# sho cos scheduler-hierarchy

class-of-service EGRESS scheduler hierarchy - rates in kbps
---------------------------------------------------------------------------------
                                    shaping guarntd delaybf  excess
interface name               index    rate    rate    rate    rate      other
---------------------------- ----- ------- ------- ------- ------- -------------
xe-2/0/0                       148       0       0       0       0
xe-2/0/1                       149       0       0       0       0
xe-2/1/0                       150       0       0       0       0
xe-2/1/1                       151       0       0       0       0
xe-2/2/0                       152  500000       0       0       0
  iflset_basic                   4    2000    1000       0       0
    xe-2/2/0.1                 328    1000     500       0       0
      q 0 - pri 0/0          20205       0      5%       0     35%
      q 1 - pri 0/0          20205       0      5%       0      5%
      q 2 - pri 0/0          20205       0     10%       0     10%
      q 3 - pri 3/1          20205       0      5%     10%      5%
      q 4 - pri 0/0          20205       0     30%       0     30%
      q 5 - pri 4/0          20205       0     30%   25000      0% exact
      q 6 - pri 0/0          20205       0     15%       0     15%
      q 7 - pri 2/5          20205       0       0       0      0%
    iflset_basic-rtp             4     500       0       0       0
      q 0 - pri 0/0          20205       0      5%       0     35%
      q 1 - pri 0/0          20205       0      5%       0      5%
      q 2 - pri 0/0          20205       0     10%       0     10%
      q 3 - pri 3/1          20205       0      5%     10%      5%
      q 4 - pri 0/0          20205       0     30%       0     30%
      q 5 - pri 4/0          20205       0     30%   25000      0% exact
      q 6 - pri 0/0          20205       0     15%       0     15%
      q 7 - pri 2/5          20205       0       0       0      0%
```

   a. H-CoS is in effect
   b. Two IFL-Sets are defined
   c. The interface is in PIR mode
   d. A remaining traffic profile is attached
   e. Both A and D

12. Which is true regarding priority handling at a L2 node?
   a. GH and GM in excess of G-Rate is not demoted
   b. GL is demoted when there is a lack of G-Rate
   c. EH and EL can be promoted into GL when there is excess credit
   d. When demoting GL, the excess level (H or L) is determined by the queue level excess priority setting
   e. All of the above

13. You have a remaining profile for the IFD only. An interface is named in an IFL-Set but does not have an IFL-level TCP applied. What happens?
   a. Nothing, this is not supported. All IFLs must have a TCP applied explicitly
   b. The IFL gets the IFD-level remaining profile
   c. The IFL gets the L2 scheduling node's remaining profile
   d. The IFL gets an unpredictable level of CoS treatment; the L2 nodes RTP shows the full IFD shaping rate

14. You have a queue set to strict-high. Which is true?
   a. This queue is not subjected to priority demotion at queue level
   b. This queue is not subjected to priority demotion at node level based on G-Rate
   c. This queue is subjected to priority demotion related to per priority shaping
   d. All of the above

Chapter Review Answers

1. **Answer: B**. Currently only queuing versions of MPC support per unit and H-CoS.
2. **Answer: A**. The excess rate option is used for Trio. Excess bandwidth share is used for older IQ2 PICs on ADPCs. You can control excess share as a percentage or proportion.
3. **Answer: B**. Trio interfaces factor both Layer 2 and Layer 1 overhead into the shaping rate and display queue statistics. This includes Ethernet's interframe gap (IFG), preamble, and FCS. You can adjust the overhead values to add or subtract bytes to ensure that shaping rates do not penalize end users for additional overhead that may be added by carrier equipment, such as a DSLAM adding multiple VLAN tags.

4. **Answer: E**. While you can oversubscribe G-Rates/queue transmit rates, this is not ideal practice unless external means are in place to ensure all queues are not simultaneously active. A queue's transmit rate is considered a G-Rate, as is setting a CIR in a TCP, which is then applied to either L2 or L3 scheduling nodes.
5. **Answer: E**. All are true. See the previous note regarding overbooking G-Rates. If you want to ensure traffic discard, you can set an IFL shaping to a rate that is higher than the IFL-Set or IFD shaping rate, but again, this is not typical. The remaining construct is a critical aspect of Trio H-CoS scale as, unlike IQ2 interfaces, it permits CoS profiled for IFLs that would otherwise have no explicit CoS configuration. The remaining profile can be just a PIR/shaped rate or can include a CIR/G-Rate to provide guaranteed service levels.
6. **Answer: E. Both C and D are true**. With regards to G-Rate, GH and GM are only demotable by queues as a function of reaching their transmit rates. At a scheduler node, GL must remain demotable (which is why B is false) to the excess region in the event that the sum of GH and GM exceed the node's G-Rate. In contrast, a scheduler node can demote any G-Rate (GH, GM, or GL) as a function of a per priority shaper.
7. **Answer: E. Both A and B are correct**. In the v11.4 release, IFL-Sets are not supported for AE interfaces, but other aspects of H-CoS are. When used for H-CoS, you can configure whether each AE member gets an equal division of the AE bundle's bandwidth or if the bundle bandwidth is replicated on each member link.
8. **Answer: E**. All are ways that can be used to limit a queue's maximum rate below that of the underlying IFL's shaping rate.
9. **Answer: E. Both C and D are correct**. When a TCP references a scheduler map, it's applied at level 3 (IFL) for per unit or H-CoS, to support queues, or at level 1 (IFD) in per port mode, but again, to support queues. When applied to an internal scheduler node, or at the IFD/root level as part of H-CoS, there are no queues so a scheduler map is not used.
10. **Answer E. A, B, and C are correct**. Trio does not use dynamic memory allocation for queue buffers.
11. **Answer E. A and D are correct**. Only H-CoS supports interface sets, and only one interface set is defined. The remaining traffic profile handles traffic for IFLs that are part of the basic set when the IFL does not have a TCP attached. There is no explicit remaining IFL-Set.
12. **Answer: E**. All of the statements are true and describe current L2 node's priority demotion and promotion.
13. **Answer: D**. C is false as there is no IFL-Set level TCP and the IFD's remaining TCP is used for interfaces that are not named in any set. Testing shows the L2 node's RTP displays the IFD shaping rate typically adopts a default scheduler with 95%/5%, but the IFL gets 10 Mbps of throughput. This is not a supported configuration and one day a commit check may prevent it.

In the following, the H-CoS config is modified to remove the TCP from unit 200 while leaving it in the premium set so that it is not caught by the IFD remaining profile. Unit 100 is moved into the premium IFL-Set to keep it active, as a set must have a minimum of one member. Lastly, the premium IFL-Set has its remaining profile removed. As a result, there is no place for unit 200 to go; it has no slot in the IFL-Set it's named to once its TCP is removed, and the set's remaining profile, the normal safety net for this type of mistake, has been removed. Upon commit, the tester throughput for Ifl xe-2/2/0.200 went from the expected 3 Mbps to 10 Mbps (just why is not clear), and the scheduler hierarchy listed the 500 Mbps shaped rate of the IFD as the rate for the premium IFL set's RTP.

```
[edit]
jnpr@R4# show | compare jnx_harry/cos_case_h_cos
[edit interfaces interface-set iflset_premium interface xe-2/2/0]
+     unit 100;
[edit interfaces]
-   interface-set iflset_business {
-       interface xe-2/2/0 {
-           unit 100;
-       }
-   }
[edit class-of-service interfaces interface-set iflset_premium]
-     output-traffic-control-profile-remaining tc-iflset_premium_remain;
[edit class-of-service interfaces xe-2/2/0 unit 200]
-      output-traffic-control-profile tc-ifl_premium;
NPC2(R4 vty)# sho cos scheduler-hierarchy

class-of-service EGRESS scheduler hierarchy - rates in kbps
---------------------------------------------------------------------------------
                                    shaping guarntd delaybf  excess
interface name               index    rate    rate    rate    rate      other
---------------------------- -----  ------- ------- ------- ------- -------------
. . .
xe-2/2/0                       152   500000       0       0       0
  iflset_basic                  22     2000    1000       0       0
. . .
  iflset_premium                25    30000   20000       0       0
    xe-2/2/0.100               335     2000    1000       0       0
      q 0 - pri 0/0          20205        0      5%       0     35%
      q 1 - pri 0/0          20205        0      5%       0      5%
      q 2 - pri 0/0          20205        0     10%       0     10%
      q 3 - pri 3/1          20205        0      5%     10%      5%
      q 4 - pri 0/0          20205        0     30%       0     30%
      q 5 - pri 4/0          20205        0     30%   25000      0% exact
      q 6 - pri 0/0          20205        0     15%       0     15%
      q 7 - pri 2/5          20205        0       0       0      0%
    iflset_premium-rtp          25   500000       0       0       0
      q 0 - pri 0/1              2        0     95%     95%      0%
      q 3 - pri 0/1              2        0      5%      5%      0%
. . .
```

14. **Answer: E**. All are true. Excess-high is the same priority as high, but can never be demoted. This is why a rate limit, shaping, or filter-based policing is so important

for this priority setting; otherwise, it can starve all others. In testing, it was found that a SH queue could be set to excess-priority none, and that this limited its throughput to the priority shaping rate, so that SH is demotable at nodes, and that excess none is supported for SH, at least in v11.4R1.

CHAPTER 6

MX Virtual Chassis

As the number of devices in your network grow, the operational burden increases. A common term in the networking industry is "stacking," which is a concept where multiple devices can be joined together and managed as a single device. A common problem with stacking is the lack of intelligence in the implementation, which leads to high availability problems. All too often, vendors implement a stacking solution that simply designates master and slave devices to provide bare minimum functionality. In simple stacking implementations, the failure of a networking device requires a lengthy mastership election process and no synchronization of kernel state such as routing protocol adjacencies, routing tables, and MAC address tables.

Virtual chassis really shows off Juniper's engineering prowess. When looking at a standalone chassis, there are many things that are a given: dual routing engines, nonstop routing, nonstop bridging, and graceful routing engine switchover. Virtual chassis was designed from the ground up to include these critical features and provide a true, single virtual chassis.

Why should you accept anything less? To simply refer to Virtual chassis as "stacking" is an insult.

What is Virtual Chassis?

Virtual chassis is very similar to a distributing computing concept called a single system image (SSI), which is a cluster of devices that appears to be a single device. However, simply appearing to be a single device isn't good enough; all of the fault tolerance features that are available in a physical chassis need to be present in the virtual chassis as well. This creates a unique engineering challenge of constructing a virtual chassis that looks, feels, and behaves like a true chassis, as shown in Figure 6-1. The two routers R1 and R2 are joined together by a virtual chassis port (VCP) to form a virtual chassis.

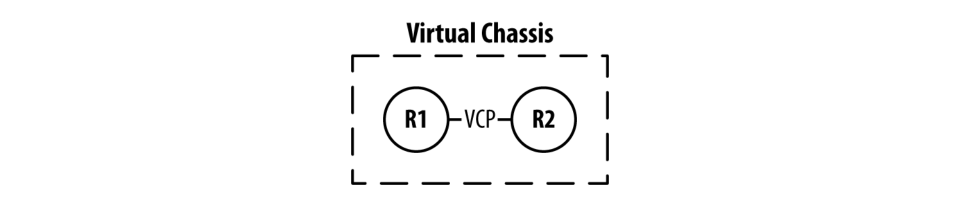


Figure 6-1. Illustration of Virtual Chassis.

Once the two routers have been configured to participate in virtual chassis, the virtual chassis will now act as a single router. For example, when you log in to the router and execute commands such as `show chassis hardware` or `show interfaces terse`, you will see the hardware inventory and interface list of both routers.

There are several features that contribute to the high availability in a typical chassis:

- Redundant power supplies
- Dual routing engines
- Nonstop routing
- Nonstop bridging
- Graceful routing engine switchover
- Multiple line cards

Each component is fully redundant so that there isn't a single point of failure within a single chassis. Virtual chassis takes the same high-availability features that are found within a single chassis and extends them into multiple chassis.

Consolidating multiple chassis into a single virtual chassis creates a distinct operational advantage. Operational and business support systems (OSS/BSS) are easier to configure with virtual chassis, because although there are multiple chassis, virtual chassis looks and feels as a single chassis. The following examples are features and services that operate as a single pane of glass in virtual chassis:

- Simple Network Management Protocol (SNMP)
- Authentication, Authorization, and Accounting (AAA); this includes services such as RADIUS and TACACS+
- Junos Application Programming Interface (API)
- Secure Shell (SSH)
- Configuration management
- Routing and switching

Virtual chassis is able to operate as a single entity because a single routing engine has ownership of all of the data planes across all chassis within the virtual chassis.

MX-VC Terminology

Virtual chassis introduces a lot of new concepts and terminology. Let's begin by starting with the basics of virtual chassis and review the components and names.

VC
: Virtual chassis represents the SSI of all the physical chassis combined.

VCCP
: Virtual Chassis Control Protocol is a Juniper proprietary protocol that's based on IS-IS to discover neighbors and build a virtual chassis topology.

VCP
: VC Port. Every chassis in a virtual chassis requires some sort of data and control plane connectivity for VCCP to operate. Revenue ports are reserved in advanced and transformed into VCP interfaces that are dedicated exclusively for interchassis data plane and VCCP control traffic.

There is also a set of virtual chassis terminology associated with the routing engines of each component. Because each physical chassis has its own set of routing engines, there needs to be a standard method of identifying each routing engine within a virtual chassis.

Table 6-1. Virtual Chassis Routing Engine Terminology.

Term	Definition
VC-M	Virtual Chassis Master
VC-B	Virtual Chassis Backup
VC-L	Virtual Chassis Line Card
VC-Mm	Master Routing Engine in VC-M
VC-Mb	Backup Routing Engine in VC-M
VC-Bm	Master Routing Engine in VC-B
VC-Bb	Backup Routing Engine in VC-B
VC-Lm	Master Routing Engine in VC-L
VC-Lb	Backup Routing Engine in VC-L

The three main components are the VC-M, VC-B, and VC-L. The other subcomponents are merely master or backup routing engine notations. Only one chassis can be the Virtual Chassis Master at any given time. A different chassis can only be the Virtual Chassis Backup at any given time. All other remaining chassis in the virtual chassis are referred to as Virtual Chassis Line Cards. Let's put all of the virtual chassis pieces together, as shown in Figure 6-2.

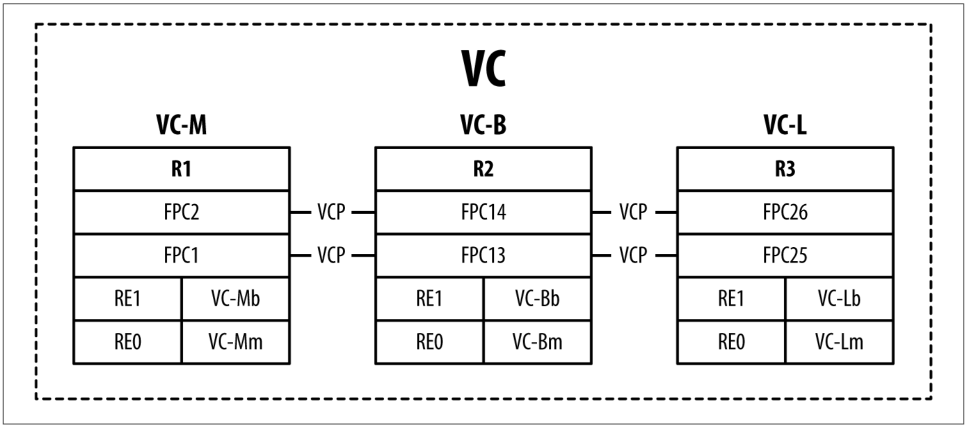


Figure 6-2. Illustration of Virtual Chassis Components.

At a high level, there is a single virtual chassis called `VC` that is comprised of three physical chassis: `R1`, `R2`, and `R3`. The chassis `R1` is the virtual chassis master (VC-M), `R2` is the virtual chassis backup (VC-B), and `R3` is a virtual chassis line card (VC-L). Each chassis has redundant routing engines; for example, RE0 on `R1` is the VC-M master routing engine whereas RE1 is the VC-M backup routing engine. The other chassis `R2` and `R3` follow the same routing engine master and backup standard except that `R2` is the VC-B and `R3` is a VC-L. Each chassis is connected to another chassis via a VC Port (VCP). The VCP is used for VCCP control traffic and a data plane for transit traffic. For example, if ingress traffic on `R1` needed to be switched to an egress port on `R3`, the traffic would traverse the VCP links from `R1` to `R2` and then finally to `R2` to `R3`.

MX-VC Use Case

Virtual chassis is designed to solve the problem of reducing the overhead of OSS/BSS by creating a single virtual chassis as the number of networking devices increase. This creates some interesting side effects that are also beneficial to the architecture of the network:

IEEE 802.3ad

By combing multiple systems into a single logical system, the ability to have node-level redundancy with IEEE 802.3ad becomes trivial. MX-VC provides node-level redundancy to any downstream IEEE 802.3ad clients and removes the requirement for spanning tree.

Because the MX-VC uses VCCP to move traffic between the members in the virtual chassis, traditional routing protocols are no longer required inside the virtual chassis. From the point of view of a packet traveling through a virtual chassis, the ingress and egress ports are on the same system and the packet's TTL is decremented.

Rapid Deployment

The single control plane of MX-VC allows the virtual chassis to grow in the number of members without downtime to the hardware or services. Additional chassis become "plug and play" from the point of view of the control plane.

It's common to see MX-VC in the core and aggregation of large data centers and mobile backhaul. Virtual chassis simplifies the logical architecture while at the same time providing physical redundancy. Layer 2 aggregation with IEEE 802.3ad works very well with virtual chassis because any downstream device will see the virtual chassis as a single logical link that prevents the possibility of a loop in the network. Spanning tree will not block the single logical link and the full bandwidth will be available for use.

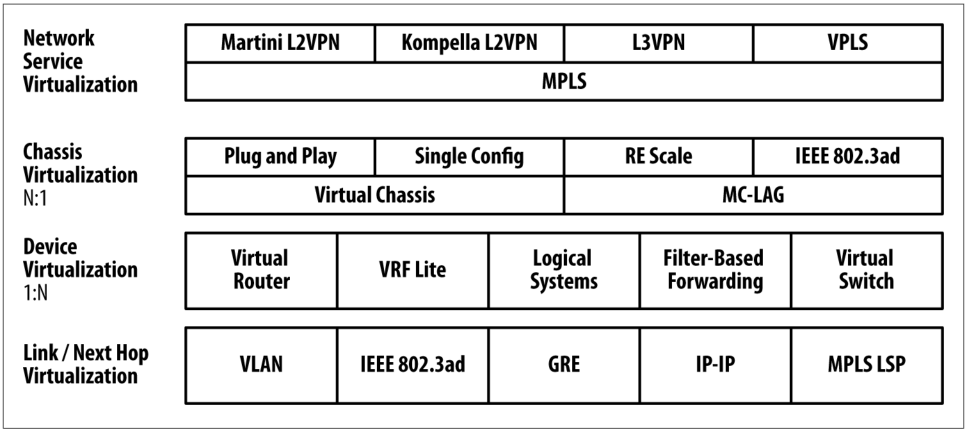


Figure 6-3. Illustration of Different Types of Juniper MX Virtualization.

Starting at the bottom of the virtualization architecture stack in Figure 6-3, the Juniper MX allows the virtualization of the links and next-hops. The next type of virtualization is the division of a single chassis that is referred to as 1:N virtualization because a single chassis is divided into N services (e.g., a router can have multiple routing instances). A level above this type of single chassis virtualization is the concept of creating network services across multiple chassis; this type of virtualization is referred to as N:1 virtualization because it takes N chassis and spreads the service across them. The final form of virtualization that can exist on top of N:1, 1:N, or simple next-hop virtualization is the concept of network service virtualization. Technologies such as MPLS are able to create virtual private networks (VPN) based on L2 or L3 network services in the form of point-to-point or point-to-multipoint.

MX-VC Requirements

There are only a few requirements to be able to use virtual chassis on the Juniper MX. The goal of virtual chassis is to create highly resilient network services, and this requires the following:

Trio-based line cards
: Virtual chassis can only be supported with MPC family line cards that utilize the Trio chipset. Older-generation DPC line cards will not be supported.

Dual Routing Engines
: As of Junos v11.4, the maximum number of members within a virtual chassis is two. When two chassis are configured for virtual chassis, each chassis must have dual routing engines. As the number of members in a virtual chassis is increased in the future, the requirement for dual routing engines per chassis maybe eliminated. The same type of routing engine must be used in all chassis as well. This is because of the strict requirements of GRES, NSR, and NSB synchronization. For example, if the RE-1800x4 is used in member0, the RE-2000 cannot be used in member1.

Latency
: It is possible to create a virtual chassis that spans cages in a data center, wiring closets, or any distance that provides less than 100 ms of latency. This allows for some creative solutions. The latency requirement is imposed on the VCP interfaces. For example, using ZR optics, it's possible to directly connect the chassis up to 80 kilometers away.

Junos Version
: The version of Junos on each routing engine must be identical or virtual chassis will not come up correctly.

VCP Interface Speed
: Either 1G (ge) or 10G (xe) interfaces can be used to configure VCP interfaces. The only restriction is that both types of interfaces cannot be used at the same time. It's recommended that 10G (xe) interfaces be used in the configuration of VCP interfaces. The rule of thumb is to calculate the aggregate throughput of the virtual chassis and configure 50% of that bandwidth as VCP interfaces. For example, if a virtual chassis would be forwarding an aggregate 100 Gbps of traffic, it's recommended that at least five 10G interfaces be configured as VCP interfaces.

IEEE 802.1Q on VCP Interfaces
: Typically, VCP interfaces are directly connected between members within a virtual chassis; however, if you need to use an intermediate switch between members, the VLAN ID used by the VCP interfaces is 4,094.

The MX-VC hardware requirements are a bit different from the EX. For example, the EX4200 and EX4500 use a special VCP interface on the rear of the switch, whereas the Juniper MX doesn't require any special VCP hardware; only regular revenue ports on MPC line cards are required for VCP interfaces. Another example is that the EX8200 requires a special external routing engine to create a virtual chassis; however, the MX doesn't require any special routing engine. The only routing engine requirement is that all of the routing engines in the virtual chassis must be the same model and run the same version of Junos.

MX-VC Architecture

Multiple chassis working together to emulate a single virtual chassis is such a simple yet elegant concept. Virtual chassis opens the door to new architectural designs that were previously impossible. For example, Figure 6-4 illustrates that `R1` and `R2` are now able to provide an IEEE 802.3ad link to both `S1` and `S2`. From the point of view of `S1` and `S2`, the router on the other end of the IEEE 802.3ad link appears to be a single entity. Cross-chassis IEEE 802.3ad requires no special configuration or knowledge on `S1` and `S2`. All of the magic happens on `R1` and `R2` using the VCCP protocol to appear as a single virtual chassis.

Another benefit of virtual chassis is being able to simplify the OSS/BSS. Because the multiple chassis now have a single control plane, services such as RADIUS, SNMP, and syslog only need to be configured and installed once. For example, billing systems can leverage SNMP to look at traffic counters for the virtual chassis as a whole; it appears to be one large router with many different interfaces. Another example is that any sort of alerts, messages, or failures will be handled by a single routing engine on the VC-M and can be processed via syslog.

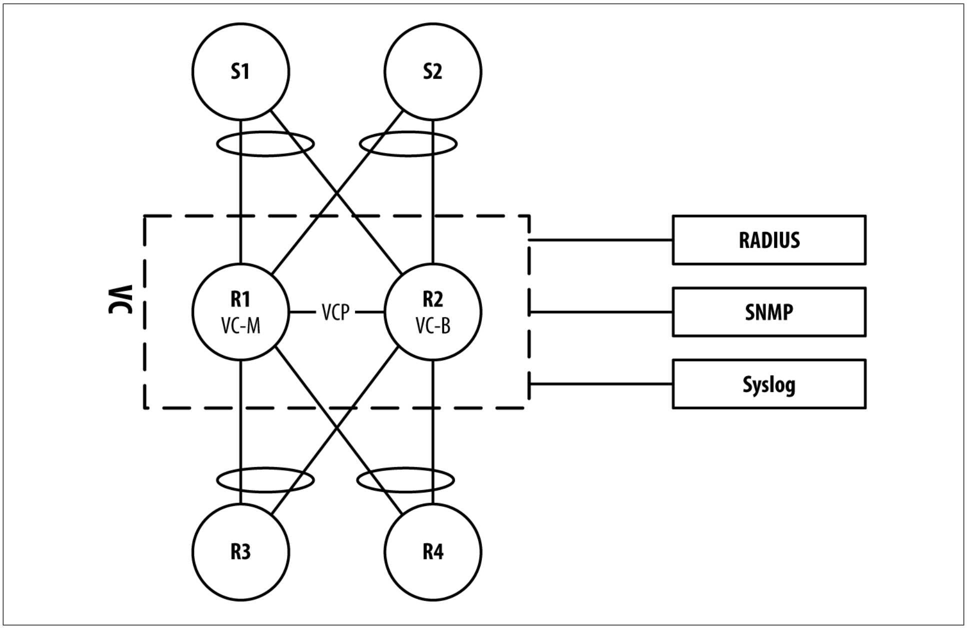


Figure 6-4. Illustration of Virtual Chassis Concept.

Traditionally, a single chassis has two routing engines: one master and one backup. As described in Chapter 9, the backup routing engine runs a `ksyncd` process that keeps itself up to date with kernel runtime objects. Through this synchronization, the backup routing engine is able to takeover the control plane if the master routing engine were to fail.

Virtual chassis changes the routing engine failover architecture from a localized concept to a cross-chassis concept. For example, in Figure 6-4, the master routing engine lives in `R1` whereas the backup routing engine lives in `R2`. Just as before, there is a `ksyncd` process living on `R2` that is keeping the kernel synchronized; in the event of a failure on `R1` (VC-M), the router `R2` (VC-B) would take over as the master routing engine.

It would perhaps seem logical that if the VC-Mm were to fail the VC-Mb would take over, but this isn't the case. Virtual chassis needs to assume the worst and has to operate within the context of multiple chassis; the best course of action is to provide GRES and NSR functionality between chassis instead of within a single chassis. It's always in the best interest of virtual chassis to mitigate any single point of failure.

MX-VC Kernel Synchronization

Let's explore the virtual chassis kernel synchronization in more depth. When a chassis operates in a virtual chassis mode, there is a requirement to have both *local* and *global* kernel scope; this is different from a traditional chassis where there's only a local kernel scope. A chassis operating in virtual chassis mode is no longer the center of the universe and must understand it's cooperatively working with other chassis in a confederation.

The local kernel scope handles state that's local to the chassis and isn't required to be shared with other chassis. An example of local kernel scope would be the control of the local VCP interfaces; they are set up and configured individually on each chassis without concern of other chassis in the virtual chassis. The global kernel scope is basically everything else; some examples include hardware inventory and IFL state.

Virtual chassis introduces a new concept called a relay daemon (`relayd`) which is designed to reduce the number of PFE connections to the kernel, as illustrated in Figure 6-5. The `relayd` passes Inter-Process Communication (IPC) messages between the routing engine kernel and the line card. The number of PFE connections are reduced because `relayd` acts as a proxy per chassis. For example, if VC-L had 12 line cards, each line card would have a connection per PFE to the VC-L `relayd`; in turn, the VC-L `relayd` would have a single connection to the VC-M routing engine kernel instead of 12. Each chassis has a `relayd` process on the master routing engine providing the following functions:

- Each line card has a PFE connection to the local chassis' `relayd` process for both the local and global state.
- Each chassis' master routing engine synchronizes state between the local kernel and local `relayd` state.
- If the chassis is the VC-M, `relayd` will synchronize its global state with the kernel global state.

- If the chassis the VC-B or a VC-L, relayd will synchronize its global state with the VC-Mm global kernel state.

Recall that GRES and NSR use the ksyncd process on the backup routing engine to keep state synchronized. Virtual chassis will use GRES and NSR to synchronize state from the VC-Mm to the VC-Bm, as illustrated in Figure 6-5.

- Each chassis' backup routing engine will have a ksyncd process to synchronize the local kernel state between the master and backup routing engines.
- The VC-B will have a special ksyncd process on the master routing engine to synchronize global kernel state from the VC-M master routing engine.

This architecture of kernel synchronization and delegation makes it very easy for Junos to establish a true virtual chassis that behaves just like a real traditional chassis. Each chassis will individually manage their own VCP interfaces with the local kernel state, whereas the VC-M will manage all other chassis in the virtual chassis as if it was just a simple extension of hardware.

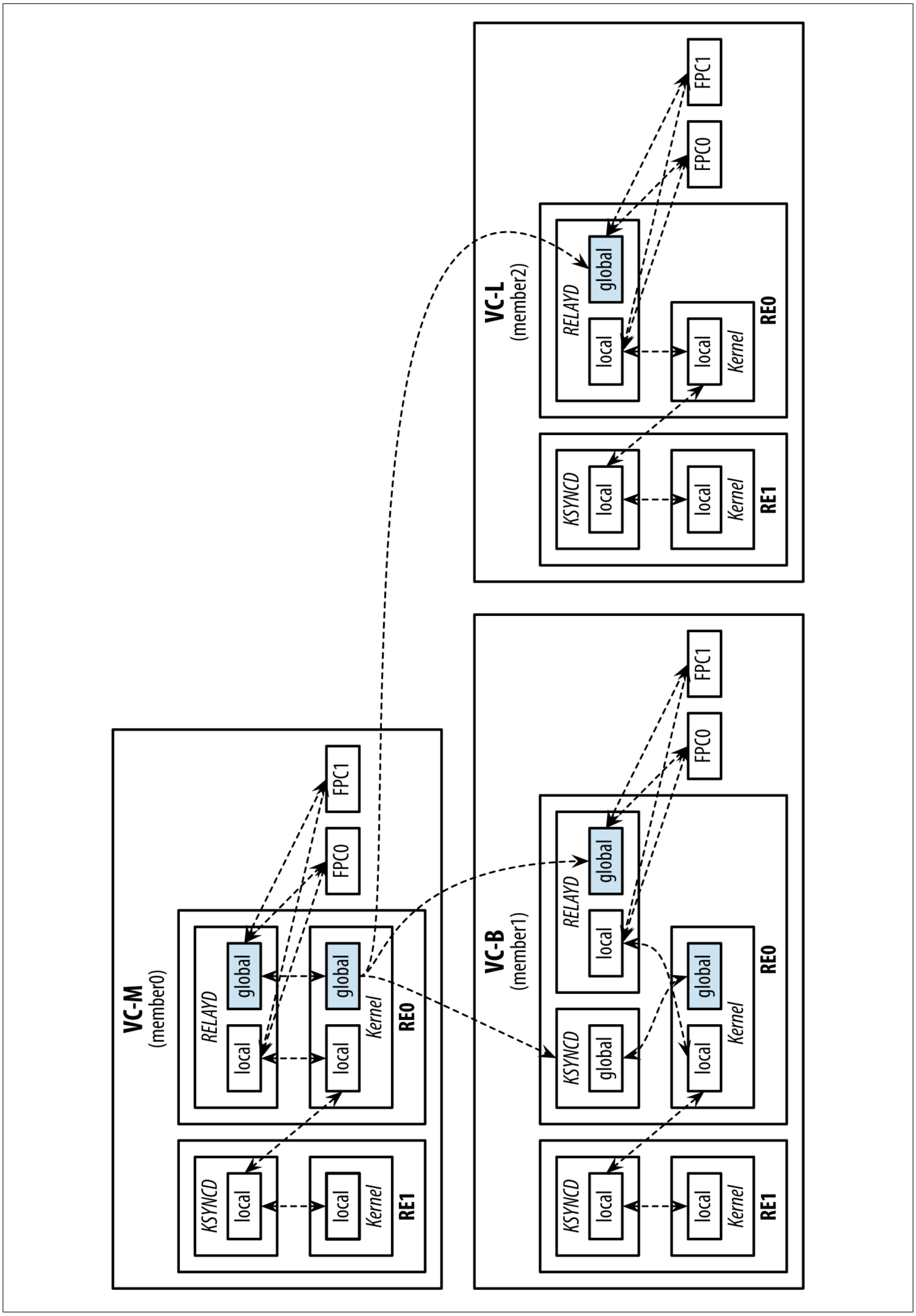


Figure 6-5. Illustration of Virtual Chassis Kernel Replication.

Let's take a quick look at the kernel processes on the VC-M and VC-B and verify the proper locations of `ksyncd` and `relayd`. It's expected that the VC-Mm will have a `relayd` process while the VC-Mb will have a `ksync` process.

Let's look at the VC-Mm first:

```
{master:member0-re0}
dhanks@R1-RE0>show system processes extensive | match relayd
18258 root        1  96    0  4528K  2620K select   0:00  0.00% relayd
```

It's confirmed that `relayd` is operating on the VC-Mm in order to synchronize global kernel state to local line cards and global kernel state to other chassis in the virtual chassis. Now let's look at the VC-Mb:

```
{master:member0-re0}
dhanks@R1-RE0>request routing-engine login re1

 --- JUNOS 11.4R2.8 built 2012-02-16 22:46:01 UTC
warning: This chassis is operating in a non-master role as part of a virtual-chassis
         (VC) system.
warning: Use of interactive commands should be limited to debugging and VC Port
         operations.
warning: Full CLI access is provided by the Virtual Chassis Master (VC-M) chassis.
warning: The VC-M can be identified through the show virtual-chassis status command
           executed at this console.
warning: Please logout and log into the VC-M to use CLI.
{local:member0-re1}
dhanks@R1-RE1>show system processes extensive | match ksyncd
 1451 root        1  96    0  5036K  2908K select   0:01  0.00% ksyncd
```

Because virtual chassis transforms a group of chassis into a single virtual chassis, you will need to use the `request routing-engine login` command to access other routing engines within the virtual chassis. As expected, the backup routing engine on `R1` (VC-Mb) has a `ksyncd` process to synchronize local kernel state.

Now that the kernel synchronization on VC-M has been verified, let's move on to the VC-B. It's expected that the VC-Bm will have a copy of both `ksyncd` and `relayd`:

```
{master:member0-re0}
dhanks@R1-RE0>request routing-engine login member 1 re0

 --- JUNOS 11.4R2.8 built 2012-02-16 22:46:01 UTC
warning: This chassis is operating in a non-master role as part of a virtual-chassis
(VC) system.
warning: Use of interactive commands should be limited to debugging and VC Port
operations.
warning: Full CLI access is provided by the Virtual Chassis Master (VC-M) chassis.
warning: The VC-M can be identified through the show virtual-chassis status command
executed at this console.
warning: Please logout and log into the VC-M to use CLI.
{backup:member1-re0}
dhanks@R2-RE0>show system processes extensive | match relayd
 1972 root        1  96    0  4532K  2624K select   0:00  0.00% relayd

{backup:member1-re0}
```

```
dhanks@R2-RE0>show system processes extensive | match ksyncd
 1983 root        1  96    0  5032K  2988K select   0:00  0.00% ksyncd
```

Using the request routing-engine login command to login the master routing engine on R2, it was evident that both relayd and ksyncd were operating. The VC-Bm uses relayd to synchronize local relayd global state to the VC-Mm kernel global state. The next function of relayd synchronizes the local state with the local kernel state. The final function of relayd on VC-Bm synchronizes both the local and global state between the local chassis line cards. Finally, ksyncd will synchronize global state with the VC-Mm global kernel state and the local chassis global kernel state.

Now let's check the VC-Bb to verify that ksyncd is up and operational:

```
{master:member0-re0}
dhanks@R1-RE0>request routing-engine login member 1 re1

 --- JUNOS 11.4R2.8 built 2012-02-16 22:46:01 UTC
warning: This chassis is operating in a non-master role as part of a virtual-
           chassis (VC) system.
warning: Use of interactive commands should be limited to debugging and VC
           Port operations.
warning: Full CLI access is provided by the Virtual Chassis Master
           (VC-M) chassis.
warning: The VC-M can be identified through the show virtual-chassis status
           command executed at this console.
warning: Please logout and log into the VC-M to use CLI.
{local:member1-re1}
dhanks@R2-RE1>show system processes extensive | match ksyncd
 1427 root        1  96    0  5036K  2896K select   0:00  0.00% ksyncd
```

As suspected, ksyncd is running on the backup routing engine on R2 (VC-Bb); its responsibility is to keep the local kernel state synchronized between VC-Bm and VC-Bb.

As of Junos v11.4, the MX only supports two members in a virtual chassis, so the author were unable to demonstrate the VC-L kernel synchronization.

MX-VC Routing Engine Failures

Given there are many different components that make up a virtual chassis, let's analyze each type of routing engine failure and how the virtual chassis recovers. There are six different types of routing engines in a virtual chassis:

- VC-Mm
- VC-Mb
- VC-Bm
- VC-Bb
- VC-Lm

- VC-Lb

Each routing engine failure is a bit different. Let's walk through all of them one at a time and observe the before and after failure scenarios.

VC-Mm failure. The failure of the VC-Mm will trigger a GRES event and the VC-B will become the new VC-M. Let's take a look:

Recall that virtual chassis performs global kernel state replication between the VC-Mm and the VC-Bm. In the event that VC-Mm fails, the only other routing engine in the virtual chassis that's capable of performing a GRES would be the VC-Bm. After the failure, the VC-Mb becomes the new VC-Bb, and both routing engines in the VC-B become the new VC-M.

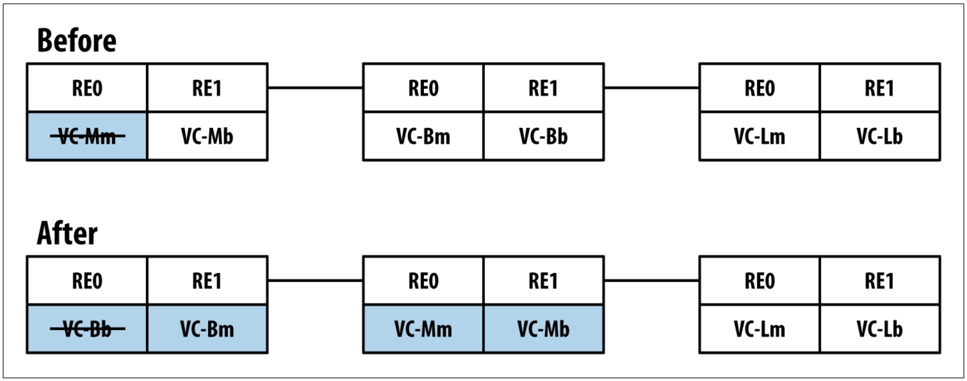


Figure 6-6. Illustration of MX-VC VC-Mm Failure.

Let's take a look at the VC-Mm before the failure:

```
{master:member0-re0}
dhanks@R1-RE0>show task replication
        Stateful Replication: Enabled
        RE mode: Master

    Protocol                Synchronization Status
    IS-IS                   Complete
```

The protocol synchronization for IS-IS is complete. Let's trigger a failover on VC-Mm:

```
{master:member0-re0}
dhanks@R1-RE0>request chassis routing-engine master switch
Toggle mastership between routing engines ? [yes,no] (no) yes

Resolving mastership...
Complete. The other routing engine becomes the master.

{local:member0-re0}
dhanks@R1-RE0>
```

At this point, the VC-Mm has become the new VC-Bb and the VC-Mb has become the new VC-Bm. In addition, the VC-B has become the new VC-M. Let's verify the new VC-Mm:

```
{local:member0-re0}
dhanks@R1-RE0>request routing-engine login member 1 re0

 --- JUNOS 11.4R2.8 built 2012-02-16 22:46:01 UTC
{master:member1-re0}
dhanks@R2-RE0> show task replication
        Stateful Replication: Enabled
        RE mode: Master

    Protocol                Synchronization Status
    IS-IS                   Complete
```

The annoying warning banner didn't show up this time because when you login to the VC-Mm, it's the master routing engine for the virtual chassis. As expected, the old VC-Bm has become the new VC-Mm and is showing the protocol synchronization for IS-IS is complete. Let's verify the switchover from the new VC-Bm:

```
{local:member0-re0}
dhanks@R1-RE0>request routing-engine login re1

 --- JUNOS 11.4R2.8 built 2012-02-16 22:46:01 UTC
warning: This chassis is operating in a non-master role as part of a virtual-
           chassis (VC) system.
warning: Use of interactive commands should be limited to debugging and VC Port
           operations.
warning: Full CLI access is provided by the Virtual Chassis Master (VC-M) chassis.
warning: The VC-M can be identified through the show virtual-chassis status command
           executed at this console.
warning: Please logout and log into the VC-M to use CLI.
{backup:member0-re1}
dhanks@R1-RE1>show system switchover
member0:
--------------------------------------------------------------------------
Graceful switchover: On
Configuration database: Ready
Kernel database: Ready
Peer state: Steady State

member1:
--------------------------------------------------------------------------
Graceful switchover is not enabled on this member
```

Everything looks great. GRES is turned on and the configuration and kernel database are ready for switchover.

VC-Mb failure. A very simple failure scenario requiring no change is the VC-Mb. There's no topology change, mastership election, or GRES switch.

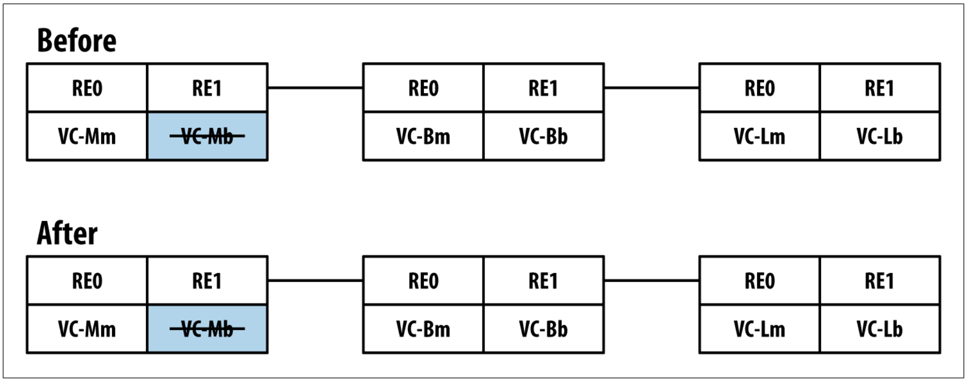


Figure 6-7. Illustration of MX-VC VC-Mb Failure.

VC-Bm failure. The next type of routing engine failure is the master routing engine on the VC-B chassis. This will cause the two routing engines on VC-B to change roles. The VC-Bb will become the new VC-Bm and vice versa.

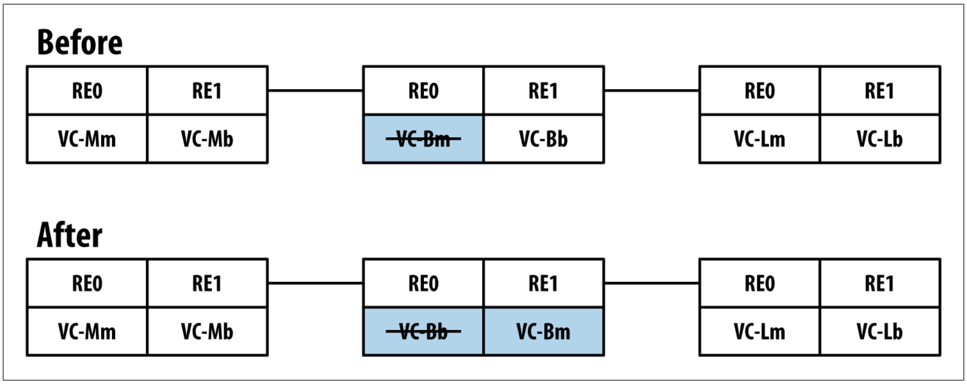


Figure 6-8. Illustration of MX-VC VC-Bm Failure.

Recall that the VC-Mm and the VC-Bm have a synchronized global kernel state; the failure of the VC-Bm will cause the GRES, NSR, and NSB replication to stop until the VC-Bb becomes the new VC-Bm and reestablishes connectivity back to the VC-Mm.

Let's take a quick look at the GRES on VC-Mm before the failure:

```
{master:member0-re0}
dhanks@R1-RE0>show task replication
        Stateful Replication: Enabled
        RE mode: Master

    Protocol                Synchronization Status
    IS-IS                   Complete
```

Now let's switch the routing engine mastership on VC-B:

```
{master:member0-re0}
dhanks@R1-RE0>request routing-engine login member 1 re0

 --- JUNOS 11.4R2.8 built 2012-02-16 22:46:01 UTC
warning: This chassis is operating in a non-master role as part of a virtual-chassis
(VC) system.
warning: Use of interactive commands should be limited to debugging and VC Port
operations.
warning: Full CLI access is provided by the Virtual Chassis Master (VC-M) chassis.
warning: The VC-M can be identified through the show virtual-chassis status command
executed at this console.
warning: Please logout and log into the VC-M to use CLI.
{backup:member1-re0}
dhanks@R2-RE0>request chassis routing-engine master release
Request the other routing engine become master ? [yes,no] (no) yes

Resolving mastership...
Complete. The other routing engine becomes the master.

{local:member1-re0}
dhanks@R2-RE0>
```

Now that the routing engines in VC-B have changed roles, let's take another look at the GRES synchronization on VC-Mm:

```
{master:member0-re0}
dhanks@R1-RE0> show task replication
        Stateful Replication: Enabled
        RE mode: Master

    Protocol                Synchronization Status
    IS-IS                   NotStarted
```

Just as expected; the GRES synchronization isn't started because of the recent routing engine switch on the VC-B. Let's wait another 30 seconds and try again:

```
{master:member0-re0}
dhanks@R1-RE0>show task replication
        Stateful Replication: Enabled
        RE mode: Master

    Protocol                Synchronization Status
    IS-IS                   Complete
```

Perfect. Now the VC-Mm is synchronized with the VC-B again. Let's check from the perspective of the new VC-Bm (RE1 on R2) to verify:

```
{master:member0-re0}
dhanks@R1-RE0>request routing-engine login member 1 re1

 --- JUNOS 11.4R2.8 built 2012-02-16 22:46:01 UTC
warning: This chassis is operating in a non-master role as part of a virtual-chassis
(VC) system.
warning: Use of interactive commands should be limited to debugging and VC Port
operations.
warning: Full CLI access is provided by the Virtual Chassis Master (VC-M) chassis.
```

```
warning: The VC-M can be identified through the show virtual-chassis status command
executed at this console.
warning: Please logout and log into the VC-M to use CLI.
{backup:member1-re1}
dhanks@R2-RE1>show system switchover
member0:
--------------------------------------------------------------------------
Graceful switchover is not enabled on this member

member1:
--------------------------------------------------------------------------
Graceful switchover: On
Configuration database: Ready
Kernel database: Ready
Peer state: Steady State
```

Everything checks out at this point. The VC-Mm protocol synchronization for IS-IS is complete, the new VC-Bm is configured for GRES, and the configuration and kernel are ready for failover.

VC-Bb failure. Another simple failure scenario is the VC-Bb. It's very similar to the VC-Mb failure scenario. There's no topology change or mastership election.

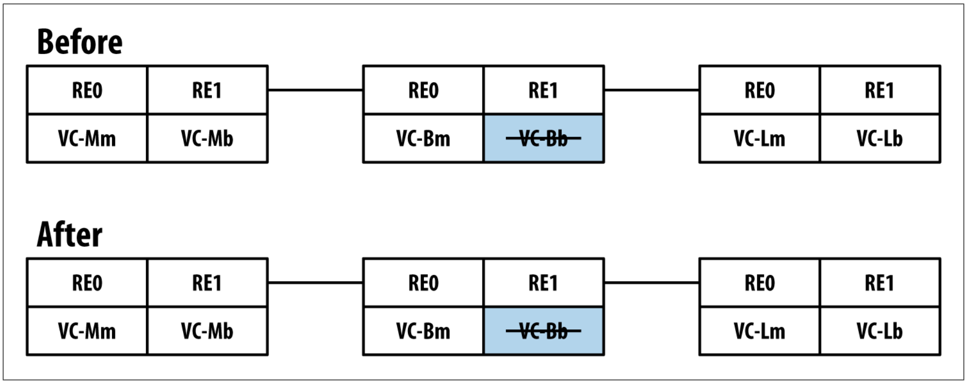


Figure 6-9. Illustration of MC-VC VC-Bb Failure.

VC-Lm failure. Virtual chassis line card failures are easier to handle because there's no global kernel synchronization or mastership election processes to deal with. In the event that a VC-Lm fails, the local backup routing engine simply takes over.

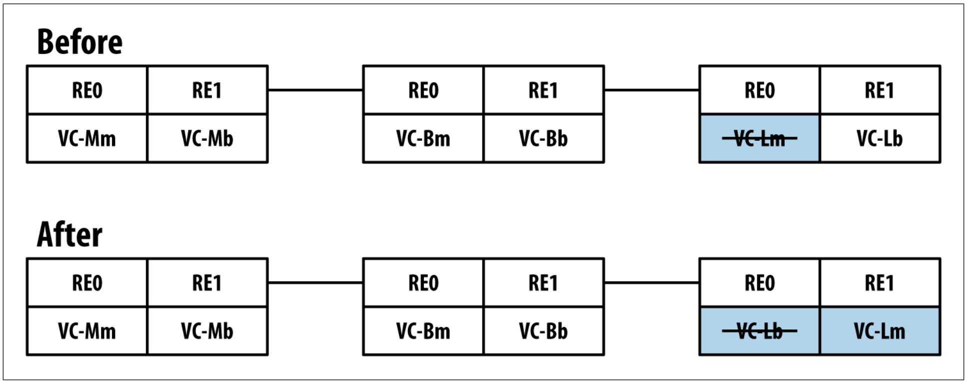


Figure 6-10. Illustration of MX-VC VC-Lm Failure.

VC-Lb. The final virtual chassis routing engine failover scenario is the VC-Lb. This is another simple failover scenario that's similar to the VC-Mb and VC-Bb. There's no topology change or mastership election process.

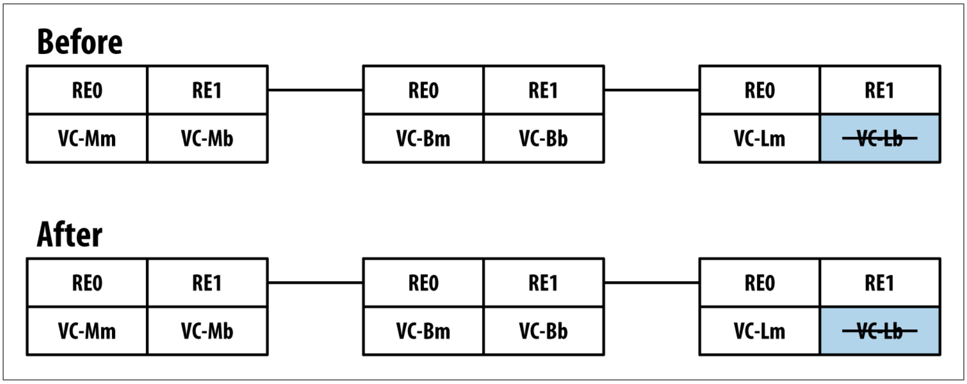


Figure 6-11. Illustration of MX-VC VC-Lb Failure.

In summary, the members of the virtual chassis work together to fully synchronize all global kernel states so that in an event of a failure, the virtual chassis can switch from the VC-Mm to the VC-Bm and reap all the benefits of GRES, NSR, and NSB. Any failure of the VC-Mm or VC-Bm requires a GRES switchover and mastership election because these two routing engines synchronize the global kernel state between the VC-M and VC-B chassis. Failures of the VC-Mb, VC-Bb, VC-Lm, and VC-Lb are very low impact and do not trigger a topology change or GRES switchover because these routing engines aren't responsible for global kernel state.

MX-VC Interface Numbering

With the introduction of virtual chassis, the interface numbering is a bit different than a traditional chassis. Recall that the kernel has been broken into two scopes: local and global. Virtual chassis requires VCP interfaces connect all chassis members in the virtual chassis. There's no requirement for special hardware when using virtual chassis on the MX platform; standard revenue ports can be converted into VCP interfaces. Each chassis' local kernel state handles VCP interfaces individually and isn't synchronized between members of the virtual chassis. Because VCP interfaces are handled by the local kernel state, the FPC numbers do not change and remain the same as if it were a single chassis. For example, in Figure 6-12, each chassis is a MX240 with two FPCs. Each chassis will define and configure a local VCP that connects to an adjacent chassis. From the perspective of each chassis, the VCP interface will live in FPC2. Figure 6-12 doesn't contain a typo; routers R2 and R3 really do say FPC13, FPC14, FPC25, and FPC26. MX-VC VCP interfaces are local kernel state and use the traditional FPC numbering scheme; however, global kernel state uses a different interface numbering system.

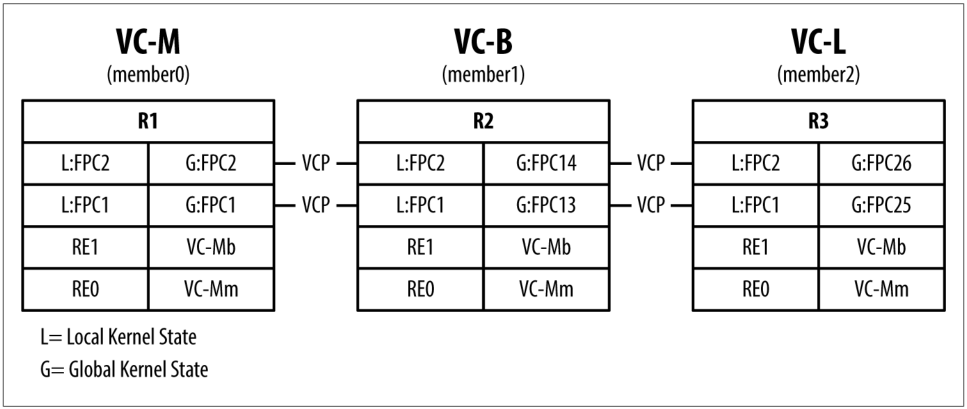


Figure 6-12. Illustration of MX-VC Interface Numbering.

The local kernel state for VCP interfaces is required because the VCP interfaces are configured and defined before the virtual chassis is up and active. Because VCP interfaces are required to bring up the VCCP protocol and form adjacencies, it's a chicken or egg problem. The easiest solution was to create VCP interfaces on each chassis using traditional interface numbering that tied to the local FPCs installed into the chassis.

The global kernel state manages everything else, including the global interface and FPC inventory and numbering scheme. The formula to calculate the MX-VC FPC is:

$$\boxed{(\text{member-id} * 12) + \text{Local FPC} = \text{MX-VC FPC}}$$

Figure 6-13. MX-VC FPC Formula.

This formula holds true for all MX platforms and doesn't change. Obviously, the base line for the MX-VC FPC formula was the MX960 because it's able to accept 12 FPCs.

So whether you configure MX-VC on the MX240, MX480, or MX960, the MX-VC FPC formula is the same. For example, in Figure 6-12, R3 has two FPCs: FPC1 and FPC2. To calculate the MX-VC FPC, simply multiply the member-id of 2 times 12 and add the local FPC number. FPC2 on R3 would become a MX-FPX of 26, as shown in Figure 6-14:

$$(2 * 12) + 2 = 26$$

Figure 6-14. Example Calculating MX-VC FPC for FPC2 on R3.

Now Figure 6-12 should come full circle. VCP interfaces use local kernel state while all other interfaces use the global kernel state. For example, assume that the VCP interface on R3 was on FPC2, PIC0, port 0. The VCP interface would be xe-2/0/0. All other ports on the same FPC and PIC would be calculated different in the context of the global kernel state for MX-VC; for example, FPC2, PIC0, port 1 on R3 would become xe-26/0/1.

MX-VC Packet Walkthrough

A packet can be forwarded locally within a single chassis or it may have to go through an intermediate member within a virtual chassis to reach its final destination. It all depends on which members within a virtual chassis hold the ingress and egress interfaces; if the ingress and egress ports are on different members, the packet must travel through the VCP interfaces that have the shortest path to the egress member. Each member within a virtual chassis has a device route table that's populated by VCCP; this device route table is used to find which member contains the forwarding information for the packet. Figure 6-15 illustrates a worst-case scenario: a packet is received on *member0* on xe-0/0/0 and needs to be forwarded through *member1* to reach the egress interface xe-36/0/0 on *member2*.

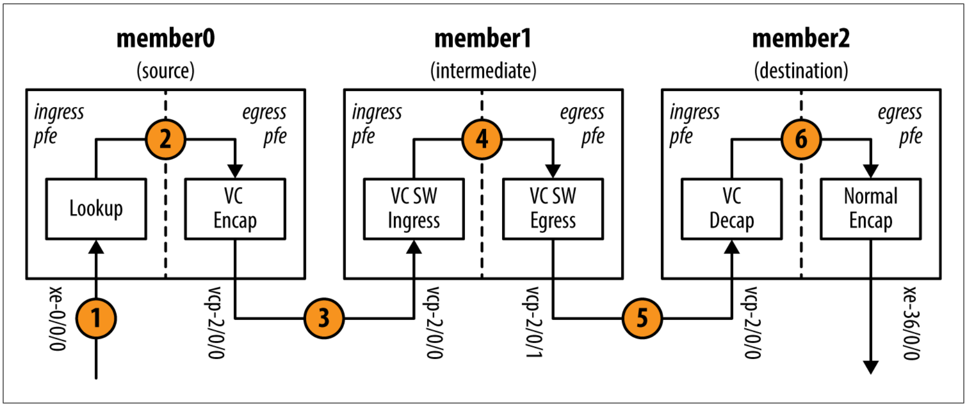


Figure 6-15. Worst-Case MX-VC Packet Forwarding Path.

In Figure 6-15, each member has both an ingress and egress PFE, and traffic needs to be forwarded across the switch fabric. There are several operations that happen at each step of the way through this example:

member0 ingress
: The lookup process gives you the global egress PFE number, which in this case is the egress PFE on member2.

member0 egress
: Find shortest path from member0 to member3. In this example, member0 has to go through member1 in order to reach the final destination of member2. Encapsulate the packet for VCP transport via vcp-2/0/0.

member1 ingress
: Do a device route lookup and see that the destination is member2.

member1 egress
: Forward the packet out interface vcp-2/0/1 to member2 on interface xe-2/0/0.

member2 ingress
: Do a device route lookup and see that the packet is destined to itself. Decapsulate the packet and send to the egress PFE.

member2 egress
: Normal forwarding processing and encapsulate the packet for egress.

As the packet travels through the virtual chassis, it incurs encapsulation and decapsulation overhead. It's always best practice to ensure that each member within a virtual chassis has a full mesh of VCP interfaces to every other member in the chassis for optimal forwarding.

Let's take a look at the packet encapsulation each step of the way through Figure 6-16:

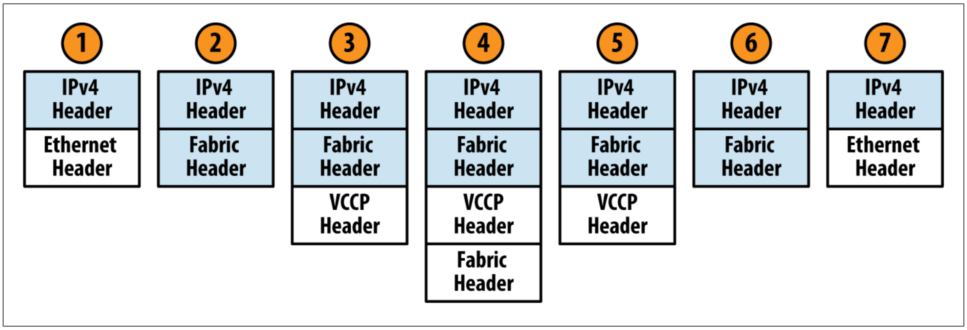


Figure 6-16. Worst-Case MX-VC Packet Encapsulation.

The numbers in Figure 6-16 correspond to the steps in Figure 6-15. As the packet makes its way through each system, it must be encapsulated for fabric or VCP and forwarded to the next-hop. The light gray boxes indicate headers that do not change as the packet is forwarded hop by hop. For example, the original IPv4 header is never changed and the fabric header that is injected via step 2 stays with the packet all the way until step 7, where it is removed. Step 4 clearly has the most overhead as it requires the original IPv4 payload, egress PFE fabric header, VCCP header, and the internal fabric header within member1 to bridge the packet from its ingress to egress PFE.

It's important to note this example is the worst-case scenario for packet forwarding within virtual chassis. To avoid forwarding packets through intermediate members within a virtual chassis, make sure there is a full mesh of VCP interface between each member in a virtual chassis, as illustrated in Figure 6-17.

The full mesh ensures that each member within the virtual chassis has a direct VCP interface to the destination member, eliminating the need for an intermediate chassis in the forwarding path. For example, the virtual chassis member `R1` has a direct VCP interface to `R2`, `R3`, and `R4`.

The VCP interfaces add an extra 42 bytes to the packet size. In the case of the MX, the maximum configurable MTU is 9,192 as of Junos 11.4. This means that packets larger than 9,150 bytes will be discarded. If you suspect traffic is being discarded because the payload is too large, check the first egress VCP interface, as this is where the traffic would be discarded.

Virtual Chassis Topology

Because VCCP is based on IS-IS, virtual chassis is able to support any type of topology. There are no restrictions on how members are connected to each other via VCP interfaces. Although a full mesh of VCP interfaces between all members in a virtual chassis, as shown in Figure 6-17, is recommended, sometimes it isn't possible.

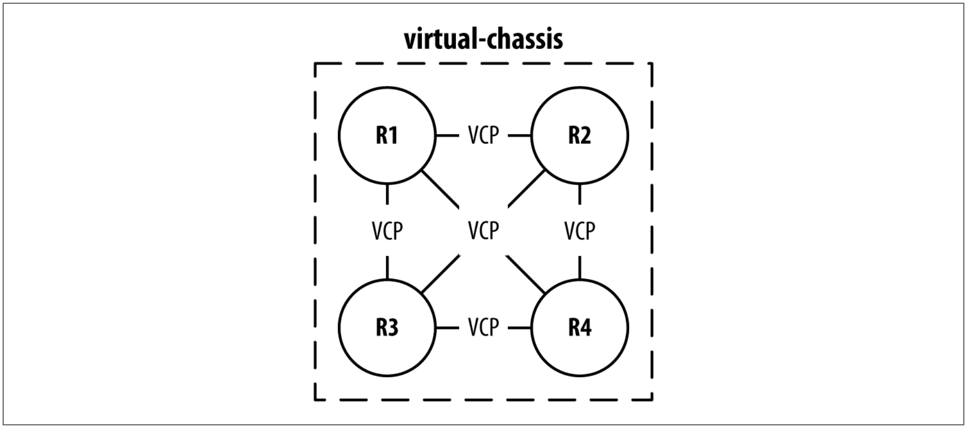


Figure 6-17. Illustration of a Full Mesh of VCP Links Between All Members Within a Virtual Chassis.

Depending on the use case, it may be more efficient to use alternative topologies in Figure 6-18 such as a ring or hub and spoke topology. For example, if the members are geographically diverse, the physical cabling may only allow a ring topology or a partial-mesh topology.

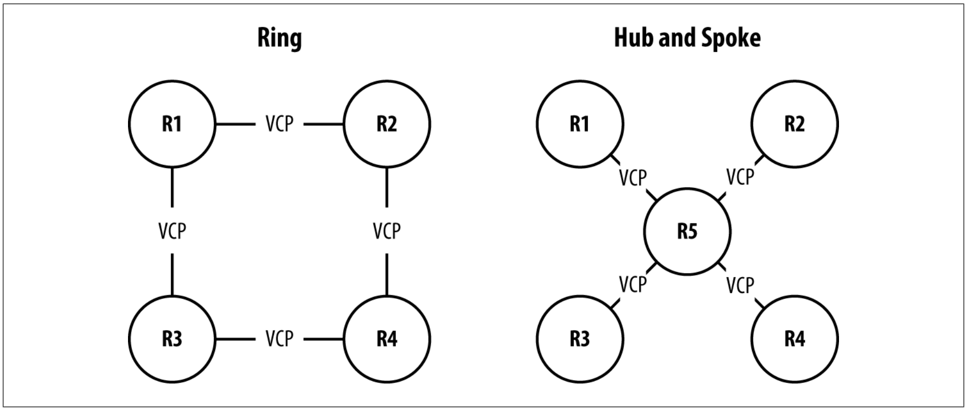


Figure 6-18. Illustration of Alternative Virtual Chassis Topologies.

Mastership Election

VCCP uses a simple algorithm to decide which member of a virtual chassis is elected as the VC-M. The mastership election process is as follows:

- Internal member priority, routing engine is set to 129, line card is set to 1, and member with undefined role is set to 128
- Prefer member of larger VC over that of smaller VC
- Prefer the current master
- Prefer the current backup
- Choose the one which has been up longer
- Choose previous master
- Choose member with lowest MAC address

The mastership election process is run every time there is a change in the VCCP topology; this could include any of the following events:

- Adding a new member from the virtual chassis
- Removing a member from the virtual chassis
- Chassis failure
- Failure of the VC-Mm
- Failure of the VC-Bm

The general rule of thumb is to choose two members in the virtual chassis to handle the VC-M responsibility and let VCCP choose which member is the VC-M. Having a particular member in the virtual chassis as the VC-M doesn't impact the transit traffic; the impact of the VC-M is only in the control plane.

Summary

Virtual chassis is a very exciting technology as it takes all of the great high-availability features from a traditional chassis such as GRES, NSR, and NSB and applies the same methodology across multiple chassis to form one, single virtual chassis. To simply refer to virtual chassis as "stacking" is an insult, because virtual chassis retains all of the control plane high-availability features and engineering, but simply spreads it across multiple chassis. Many vanilla "stacking" implementations merely attempt to give the user a single command-line experience, but when it comes to high-availability features and engineering, vanilla "stacking" fails to deliver.

To help support the high-availability features, the way kernel synchronization was performed had to be rethought. Routing engines are no longer within the same chassis being synchronized; with virtual chassis, the master and backup routing engines are in different chassis. This creates unique scaling challenges; to solve this problem, Juniper created the relay daemon to act as a proxy between FPCs and the kernel. The other challenge is that there is local state that's only relevant on a per chassis basis, while there's global state that needs to be replicated throughout the entire virtual chassis. The kernel state was separated into local and global state to solve this problem. Local state, such as local VCP interfaces, are not replicated throughout the virtual chassis and are kept local to each chassis; however, global state, such as every other FPC, will be synchronized throughout the virtual chassis.

The packet forwarding path through a virtual chassis depends on the number of intermediate members it must pass through. The majority of the time, the packet will be forwarded locally within the chassis and not incur any additional processing. The other option is that the packet needs to be forwarded to another member in the virtual chassis for processing. This requires that the packet be encapsulated and decapsulated as it moves through VCP interfaces. The worst-case scenario is that the packet must be forwarded through an intermediate member within the virtual chassis to reach its final destination. To avoid this scenario, it's recommended to create a full mesh of VCP interfaces between each member in the virtual chassis. Although there is additional encapsulation that must be performed when forwarding packets through the virtual chassis, it's important to remember that the processing is performed in hardware with one pass through the Lookup Block; the amount of processing delay is very minimal.

Just a reminder that this chapter focuses on how virtual chassis is implemented on the Juniper MX and keeps the content highly technical and assumes you already know the basics. For more information about virtual chassis, please check out *Junos Enterprise Switching* by Doug Marschke and Harry Reynolds (O'Reilly).

MX-VC Configuration

The configuration of virtual chassis on the MX is very similar to the EX. The most challenging part is collecting the required information before creating the configuration. Each chassis that is to be part the virtual chassis must be identified, and a handful of information must be collected.

Be sure to use the console port on the routing engines as you configure virtual chassis; this will ensure that if a mistake is made, the connection to the router will not be lost.

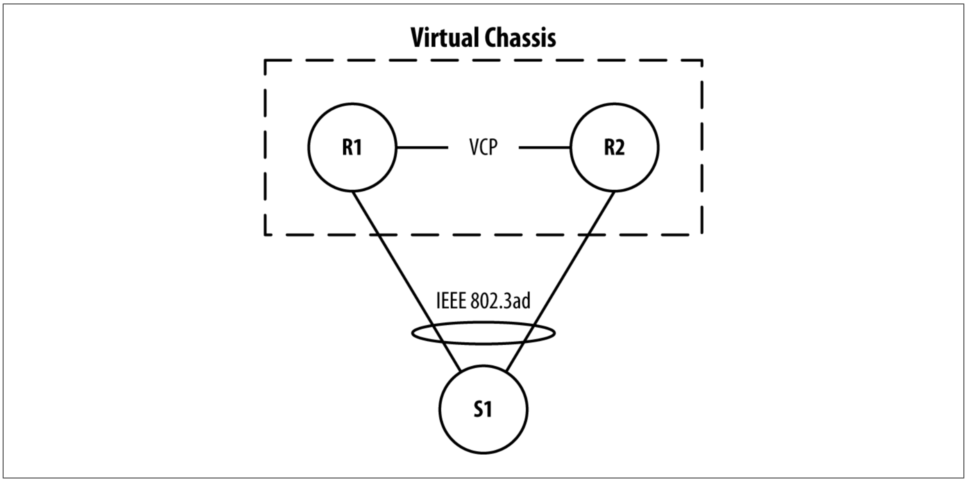


Figure 6-19. Illustration of MX-VC Configuration Topology.

This section will build a MX-VC, as illustrated in Figure 6-19, using `R1` and `R2`. The `S1` switch will be used to create an IEEE 802.3ad interface into the MX-VC.

Chassis Serial Number

Virtual chassis uses the serial number to uniquely identify each chassis in the topology. It's important to note that only the chassis serial number should be used. It's easy to confuse other serial numbers such as a power supply or line card. Use the `show chassis hardware` command to find the chassis serial number. Let's find the serial number for the `R1` chassis:

```
dhanks@R1-RE0>show chassis hardware
Hardware inventory:
Item             Version  Part number Serial number  Description
Chassis                               JN111992BAFC   MX240
Midplane         REV 07   760-021404  TR5026         MX240 Backplane
FPM Board        REV 03   760-021392  KE2411         Front Panel Display
PEM 0            Rev 02   740-017343  QCS0748A002    DC Power Entry Module
```

```
8    Routing Engine 0 REV 07   740-013063  1000745244     RE-S-2000
9    Routing Engine 1 REV 07   740-013063  9009005669     RE-S-2000
10    CB 0             REV 03   710-021523  KH6172         MX SCB
11    CB 1             REV 10   710-021523  ABBM2781       MX SCB
12    FPC 2            REV 15   750-031088  YR7184         MPC Type 2 3D Q
13      CPU
14    Fan Tray 0       REV 01   710-030216  XS7839         Enhanced Fan Tray
```

Line 4 shows the chassis serial number for the MX240 used in this book's laboratory. The chassis serial number for `R1` is `JN111992BAFC`. Now let's find the chassis serial number for `R2`:

```
1    dhanks@R2-RE0>show chassis hardware
2    Hardware inventory:
3    Item             Version  Part number Serial number Description
4    Chassis                               JN111C0B4AFC  MX240
5    Midplane         REV 07   760-021404  TR4825        MX240 Backplane
6    FPM Board        REV 03   760-021392  KE7780        Front Panel Display
7    PEM 0            Rev 02   740-017343  QCS0812A061   DC Power Entry Module
8    Routing Engine 0 REV 06   740-013063  1000690737    RE-S-2000
9    Routing Engine 1 REV 07   740-013063  1000738188    RE-S-2000
10    CB 0             REV 03   710-021523  KH6173        MX SCB
11    CB 1             REV 03   710-021523  KH3620        MX SCBf
```

The chassis serial number for `R2` is `JN111C0B4AFC` as shown in line 4. Let's save the chassis serial numbers for `R1` and `R2` and move on to the next section.

> Please note that the chassis serial number will be anchored to the routing engine located in */etc/vchassis/*. Routing engines from one chassis cannot be moved to other chassis in a virtual chassis, otherwise the virtual chassis configuration will be invalidated and you will have to start the configuration process all over again.

Member ID

Each member in a virtual chassis requires a unique member ID. Valid member IDs are 0 through 2 as of Junos 11.4. In this configuration example, the member ID for `R1` will be 0 and `R2` will use a member ID of 1. The member ID is set from the operational mode command line and will require a reboot of both routing engines. Use the **`request virtual-chassis member-id set`** command on the master routing engine to set the virtual chassis member ID. Let's configure `R1` with a member ID of 0:

```
dhanks@R1-RE0>request virtual-chassis member-id set member 0
This command will enable virtual-chassis mode and reboot the system.
 Continue? [yes,no] (no) yes

Updating VC configuration and rebooting system, please wait...

{master}
dhanks@R1-RE0>
*** FINAL System shutdown message from dhanks@R1-RE0 ***
```

```
System going down IMMEDIATELY
```

This will configure both the master and backup routing engines for virtual chassis mode. After a few minutes, the router will come back up ready for the next step. In the mean time, let's do the same for R2, but use a member ID of 1:

```
dhanks@R2-RE0>request virtual-chassis member-id set member 1
This command will enable virtual-chassis mode and reboot the system.
 Continue? [yes,no] (no) yes

Updating VC configuration and rebooting system, please wait...

{master}
dhanks@R2-RE0>
*** FINAL System shutdown message from dhanks@R2-RE0 ***

System going down IMMEDIATELY
```

You need to wait for both routers to reboot and become operational again before continuing to the next step.

R1 VCP Interface

A pair of VCP interfaces are required for R1 and R2 to build a VCCP adjacency and bring the two routers together. Each router will use the interface xe-2/0/0 as the dedicated VCP interface. Once xe-2/0/0 has been configured for a VCP interface, xe-2/0/0 will be removed from the global kernel state, renamed to interface vcp-2/0/0, and placed into the local kernel state of the chassis.

Let's begin by configuring the VCP interface as xe-2/0/0 on R1:

```
{master:member0-re0}
dhanks@R1-RE0>request virtual-chassis vc-port set fpc-slot 2 pic-slot 0 port 0
vc-port successfully set
```

Use the show virtual-chassis command to verify that the VCP interface has been successfully created:

```
{master:member0-re0}
dhanks@R1-RE0>show virtual-chassis vc-port
member0:
----------------------------------------------------------------
Interface       Type         Trunk  Status     Speed     Neighbor
or                            ID               (mbps)    ID  Interface
Slot/PIC/Port
2/0/0           Configured     3    Down       10000     1   vcp-2/0/0
```

The xe-2/0/0 interface on R1 has successfully been configured as a VCP port and is now known as vcp-2/0/0; however, the status is Down. This is because R2 needs to configure a VCP interface before the adjacency can come up.

Stop configuring VCP interfaces at this point. Do not continue to R2 and do not pass go. A global virtual chassis configuration needs to be applied on R1 before R2 is added to the virtual chassis.

It's important to stop at this point and not configure VCP interfaces on R2. There is still some global configuration work that needs to happen on R1 before R2 is brought online. This is because some of the apply-group names have changed and the virtual chassis stanza needs to be added. The process of configuring R1 with a global virtual chassis configuration first designates R1 has the VC-M of the virtual chassis.

Routing Engine Groups

The next step is to begin creating a global virtual chassis configuration on R1 before the VCP interfaces are configured on R2. One notable difference in a virtual chassis configuration is the modification in routing engine apply-groups. On a single chassis, the apply-groups re0 and re1 can be used to apply different configuration on the routing engines. With virtual chassis, the names have been changed to account for all chassis in the virtual chassis, as shown in Table 6-2.

Table 6-2. Single Chassis to Virtual Chassis Routing Engine Group Names.

		Standalone Chassis	Virtual Chassis
Router	**Routing Engine**	**Apply Group Name**	**Apply Group Name**
R1	re0	re0	member0-re0
R1	re1	re1	member0-re1
R2	re0	re0	member1-re0
R2	re1	re1	member1-re1

Note that Table 6-2 assumes that R1 has a member ID of 0 and R2 has a member ID of 1. The real apply-group name can be expressed as pseudocode where member-id is equal to the member ID of the chassis and routing-engine is equal to 0 or 1:

```
foreach $member-id (0..2)
{
    foreach $routing-engine (0..1)
    {
          printf("member%i-re%i", $member-id, $routing-engine);
    }
}
```

With this new naming method, it's possible to construct a global configuration for all routing engines within the virtual chassis using apply-groups. The first step is to copy the apply-groups re0 and re1 on R1 to member0-re0 and member0-re1:

```
master:member0-re0}
root>configure
Entering configuration mode
```

```
{master:member0-re0}[edit]
root# copy groups re0 to member0-re0

{master:member0-re0}[edit]
root# copy groups re1 to member0-re1
```

It's also possible to simply rename re0 to member0-re0 and re1 to member0-re1, but using the copy command will leave the original re0 and re1 apply-groups in place in case the virtual chassis configuration is removed in the future.

The next step is to create the routing engine apply-groups for R2; these will be named member1-re0 and member1-re1. Two methods can be used for this step. The first option is to copy the member0-re0 to member1-re0 and member0-re1 to member1-re1; however, you will have to make the necessary modifications to member1-re0 and member1-re1 such as the hostname and fxp0 IP address. The other option is to use the load merge terminal command and simply cut and paste the re0 and re1 apply-groups from R2 into R1, but don't forget to rename re0 and re1 to member1-re0 and member1-re1. The bottom line is that you need to create an apply-group for each member and routing engine in the virtual chassis. In Junos, there are many different ways to accomplish the same task, and the method you use is up to you; there's no right or wrong way.

The next step is to remove any previous routing engine apply-groups from R1:

```
{master:member0-re0}[edit]
root# delete apply-groups re0
{master:member0-re0}[edit]
root# delete apply-groups re1
```

The last step is to enable the newly created virtual chassis routing engine apply-groups on R1:

```
{master:member0-re0}[edit]
root# set apply-groups [member0-re0 member0-re1 member1-re0 member1-re1 ]
```

At this point, all standalone routing engine apply-groups have been removed and new virtual chassis routing engine apply-groups have been installed for each member and routing engine in the virtual chassis.

It's important to understand how the routing engine interfaces (`fxp0`) work in a virtual chassis. Traditionally, each routing engine has its own `fxp0` interface that can be used to directly login to a routing engine. With virtual chassis, only the VC-Mm routing engine will honor the routing engine interface. Other routing engines can configure the `fxp0` interface, but they will not respond until the virtual chassis mastership changes. For example, if the routing engine from the `member0-re0` apply-group was the current VC-Mm, it would honor the `fxp0` interface configuration in the `member0-re0` apply-group and respond, whereas the VC-Mb, VC-Bm, VC-Bb, VC-Lm, and VC-Lb routing engines would not honor their respective `fxp0` interfaces. However, if there was a topology change and `member1-re0` become the new VC-Mm, the `fxp0` configuration in the `member1-re0` apply-group would become active.

In order to log in to other routing engines, you must use the respective RS-232 console port or use the `request routing-engine login` command.

Virtual Chassis Configuration

The next step is to create the global virtual chassis configuration on `R1`. It's time to find the chassis serial numbers from `R1` and `R2`, as shown in Table 6-3.

Table 6-3. R1 and R2 Chassis Serial Numbers for MX-VC.

Router	Chassis Serial Number
`R1`	`JN111992BAFC`
`R2`	`JN111C0B4AFC`

The chassis serial numbers will be used in the preprovisioned virtual chassis configuration. In the preprovisioned virtual chassis configuration, each member ID needs to be defined along with its chassis serial number and role.

As of Junos 11.4, MX-VC requires that virtual chassis be preprovisioned.

There are two roles in virtual chassis: `routing-engine` and `line-card`. Table 6-4 illustrates that a role of `routing-engine` is able to become a VC-M, VC-B, or VC-L; however, a role of `line-card` only allows the member to become a VC-L.

Table 6-4. MX-VC Roles.

Role	Eligible MX-VC State
`routing-engine`	VC-M, VC-B, and VC-L
`line-card`	VC-L

Given that R1 and R2 will create a two-member virtual chassis, it's required that both members have a role of **routing-engine** so that they can each become a VC-M or VC-B depending on the mastership election process and failover scenarios.

Let's configure R1 with the following virtual chassis configuration:

```
virtual-chassis {
    preprovisioned;
    no-split-detection;
    member 0 {
        role routing-engine;
        serial-number JN111992BAFC;
    }
    member 1 {
        role routing-engine;
        serial-number JN111C0B4AFC;
    }
}
```

The virtual chassis configuration hard codes the serial numbers of R1 and R2 so that no other routers may inadvertently join the virtual chassis. When specifying the serial number for each member, the **preprovisioned** option is required.

GRES and NSR

The last step in the virtual chassis configuration is to ensure that GRES and NSR are configured. After all, the entire point of virtual chassis is to spread high-availability features across chassis; it would be a shame if they were forgotten about. Let's configure GRES and NSR on R1:

```
chassis {
    redundancy {
        graceful-switchover;
    }
}
routing-options {
    nonstop-routing;
}
```

Don't forget to enable commit synchronize unless you want to type commit synchronize every time you make a configuration change:

```
system {
    commit synchronize;
}
```

R2 VCP Interface

At this point, R1 has successfully been preconfigured as the first router in a virtual chassis. It's critical that the first router in a virtual chassis go through the preconfiguration checklist before adding additional chassis to the virtual chassis. Let's review the preconfiguration steps again:

Chassis Serial Number
: It's important to find and document the chassis serial number of each chassis before configuring the virtual chassis. The virtual chassis will use each chassis serial number as an anchor in the configuration. This will guarantee that only authorized chassis are granted access into the virtual chassis.

Member Role
: When designing the virtual chassis, the first decision point is to determine how the virtual chassis will handle failures. It's important to predetermine the VC-M and VC-B chassis under normal operating conditions and how the virtual chassis will react to routing engine or chassis failures.

Virtual Chassis Ports
: Each member requires a connection to other members within the virtual chassis to build the VCCP adjacency and provide a means of communication when transit data needs to be transported between members. The VCP interfaces use the local kernel state and require that you use the local chassis interface name during the configuration. It's important to remember that once an interface has been configured for VCP, that interface is no longer available for use by the system and its only purpose is for VCCP and intermember data plane traffic.

Routing Engine Apply Groups
: Virtual chassis changes the apply-groups names for the various routing engines. Instead of the traditional `re0` and `re1` apply-group names, a new naming convention is required to uniquely specify the member and routing engine within the virtual chassis. It's important to preconfigure the first member in the virtual chassis with a global apply-group configuration for all virtual chassis members and routing engines. This allows for members to be added to the virtual chassis without having to modify the configuration.

Graceful Routing Engine Switchover and Nonstop Routing
: The two core features that provide the high availability for virtual chassis are GRES and NSR. Applying the configuration on the first chassis in the virtual chassis will ensure that GRES and NSR will always be active.

Commit Synchronize
: By default, the `commit` command will only commit the configuration on the routing engine on which it was executed. To replicate the configuration change to all routing engines within the virtual chassis, the commit synchronize option must be enabled.

Now that `R1` is preconfigured and ready to accept additional members into the virtual chassis, the next logical step is to configure the VCP interface on `R2`. As soon as the VCP interface is configured on `R2`, VCCP will immediately begin to establish adjacency between `R1` and `R2` and form a virtual chassis. Once the VCCP adjacency is established, virtual chassis will perform a mastership election process. Because `R1` was configured first, the election has been rigged and `R1` will become the VC-M of the virtual chassis.

Let's now configure the VCP interface on R2 as xe-2/0/0:

```
{master:member1-re0}
dhanks@R1-RE0>request virtual-chassis vc-port set fpc-slot 2 pic-slot 0 port 0
vc-port successfully set
```

At this point, VCCP will begin establishing adjacency and forming a virtual chassis. Let's give the system a minute before checking the status.

(Wait 30 seconds.)

Let's check the VCP interfaces on R2:

```
{master:member1-re0}
dhanks@R2-RE0>show virtual-chassis vc-port
member1:
----------------------------------------------------------------
Interface       Type           Trunk  Status   Speed     Neighbor
or                              ID             (mbps)    ID  Interface
Slot/PIC/Port
2/0/0           Configured               Absent
```

By looking at the VCP status of Absent, it's obvious that VCCP still hasn't completed. Another hint is that the prompt on R2 still indicates that R2 still believes it's the master of the virtual chassis.

(Wait 30 more seconds.)

Now let's check again:

```
{backup:member1-re0}
dhanks@R2-RE0>show virtual-chassis vc-port
member0:
--------------------------------------------------------------
Interface       Type          Trunk  Status  Speed     Neighbor
or                             ID            (mbps)    ID  Interface
Slot/PIC/Port
2/0/0           Configured      3    Up      10000     1   vcp-2/0/0

member1:
--------------------------------------------------------------
Interface       Type          Trunk  Status  Speed     Neighbor
or                             ID            (mbps)    ID  Interface
Slot/PIC/Port
2/0/0           Configured      3    Up      10000     0   vcp-2/0/0
```

Very interesting; the output has changed significantly. Let's start by making a few observations. The first change is the command-line prompt; it now shows that it's the backup in the virtual chassis. The next change is that the command shows the output from both member0 and member1. Obviously, if R2 is able to get the command output from both members, virtual chassis is up and operational. The last piece of information confirming the formation of virtual chassis is the VCP status now displays Up.

Virtual Chassis Verification

Now that R1 and R2 have been configured for virtual chassis and VCCP is converged, it's time to take a closer look at the virtual chassis. The easiest method for determining the health, members, and VCP interfaces within a virtual chassis is the `show virtual-chassis status` command:

```
dhanks@R1-RE0>show virtual-chassis status

Preprovisioned Virtual Chassis
Virtual Chassis ID: 12b0.f739.21d2
  Mastership          Neighbor List
Member ID       Status Serial No    Model  priority    Role    ID  Interface
0 (FPC   0- 11) Prsnt  JN111992BAFC mx240         129  Master*  1  vcp-2/0/0
1 (FPC  12- 23) Prsnt  JN111C0B4AFC mx240         129  Backup   0  vcp-2/0/0
```

The example output provides a bird's eye view of the virtual chassis. It isn't apparent, but you can determine that both chassis are present and that R1 is currently the VC-M and R2 is the VC-B. The `show virtual-chassis status` command identifies each chassis by the serial number instead of the hostname, and the VC-M status is indicated by the role `Master` whereas the VC-B status is indicated by the role of `Backup`. There's also a helpful reminder in the second column of the command showing the FPC slot numbers.

Virtual Chassis Topology

The VCCP protocol builds a shortest path first (SPF) tree, and each node in the tree is represented by a member in the virtual chassis. Let's take a look at the VCCP adjacency:

```
{master:member0-re0}
dhanks@R1-RE0>show virtual-chassis protocol adjacency
member0:
---------------------------------------------------------------------
Interface             System         State        Hold (secs)
vcp-2/0/0.32768       001f.12b7.d800 Up                  2

member1:
---------------------------------------------------------------------
Interface             System         State        Hold (secs)
vcp-2/0/0.32768       001f.12b8.8800 Up                  2
```

VCCP has established adjacency on each member via the vcp-2/0/0 interface. Take special notice of the `System` column; it's using six octets worth of hexadecimal. It's interesting to note that another common six-octet field of hexadecimal is a MAC address. Let's take a look at the system MAC address of R1 and see if it matches the VCCP `System` value.

```
{master:member0-re0}
dhanks@R1-RE0>show chassis mac-addresses
member0:
---------------------------------------------------------------------
MAC address information:
  Public base address     00:1f:12:b8:88:00
  Public count            1984
  Private base address    00:1f:12:b8:8f:c0
  Private count           64

member1:
---------------------------------------------------------------------
MAC address information:
  Public base address     00:1f:12:b7:d8:00
  Public count            1984
  Private base address    00:1f:12:b7:df:c0
  Private count           64
```

The system MAC address of `R1` and the VCCP `System` are indeed identical. `R1` has a system MAC address of 00:1f:12:b8:88:00 and `R2` has a system MAC address of 00:1f:12:b7:d8:00. Armed with this new information, let's take a look at the VCCP route table and verify that the SPF tree is built using the system MAC address for each node.

```
{master:member0-re0}
dhanks@R1-RE0>show virtual-chassis protocol route
member0:
----------------------------------------------------------------------

Dev 001f.12b8.8800 ucast routing table             Current version: 154
----------------
System ID          Version   Metric Interface     Via
001f.12b7.d800         154        7 vcp-2/0/0.32768 001f.12b7.d800
001f.12b8.8800         154        0

Dev 001f.12b8.8800 mcast routing table             Current version: 154
----------------
System ID          Version   Metric Interface     Via
001f.12b7.d800         154
001f.12b8.8800         154          vcp-2/0/0.32768

member1:
-----------------------------------------------------------------------

Dev 001f.12b7.d800 ucast routing table             Current version: 126
----------------
System ID          Version   Metric Interface     Via
001f.12b7.d800         126        0
001f.12b8.8800         126        7 vcp-2/0/0.32768 001f.12b8.8800

Dev 001f.12b7.d800 mcast routing table             Current version: 126
----------------
System ID          Version   Metric Interface     Via
001f.12b7.d800         126          vcp-2/0/0.32768
001f.12b8.8800         126
```

From the perspective of `R1` (member0), the path to `R2` (001f.12b7.d800) has a metric of 7 via the vcp-2/0/0 interface; it also sees itself (001f.12b8.8800) in the SPF tree with a metric of 0. The same is true for `R2` (member1); it shows a path to `R1` (001f.12b8.8800) with a metric of 7 via the vcp-2/0/0 interface.

Revert to Standalone

There are two methods of deconfiguring virtual chassis: the easy way and the manual way. Each method has its own benefits and drawbacks. Let's start with the easy way. Simply login to the chassis to be removed from the virtual chassis and load the factory configuration and `commit`:

```
{master:member0-re0}[edit]
dhanks@R1-RE0# load factory-default
warning: activating factory configuration
{master:member0-re0}[edit]
dhanks@R1-RE0# commit and-quit
{master:member0-re0}
dhanks@R1-RE0>request system reboot both-routing-engines
Reboot the system ? [yes,no] (no) yes
```

After the factory default configuration has been committed, simply reboot the routing engines. Once the router reboots, it will no longer be part of the virtual chassis. The benefit is that it only requires a single command. The only downside is that the entire configuration is lost and you need to start from scratch; however, this method is the most recommended.

The other method is to execute several commands manually. The following components will need to be removed from the configuration: routing engine apply-groups and the `virtual-chassis` stanza.

```
{master:member0-re0}[edit]
dhanks@R1-RE0# delete groups member0-re0
{master:member0-re0}[edit]
dhanks@R1-RE0# delete groups member0-re1
{master:member0-re0}[edit]
dhanks@R1-RE0# delete groups member1-re0
{master:member0-re0}[edit]
dhanks@R1-RE0# delete groups member1-re1
{master:member0-re0}[edit]
dhanks@R1-RE0# delete apply-groups member0-re0
{master:member0-re0}[edit]
dhanks@R1-RE0# delete apply-groups member0-re1
{master:member0-re0}[edit]
dhanks@R1-RE0# delete apply-groups member1-re0
{master:member0-re0}[edit]
dhanks@R1-RE0# delete apply-groups member1-re1
{master:member0-re0}[edit]
dhanks@R1-RE0# delete virtual-chassis
{master:member0-re0}[edit]
dhanks@R1-RE0# commit and-quit
```

The next step is to remove the VCP interfaces:

```
dhanks@R1-RE0>request virtual-chassis member-id delete
This command will disable virtual-chassis mode and reboot the system.
 Continue? [yes,no] (no) yes

Updating VC configuration and rebooting system, please wait...
```

```
{master}
dhanks@R1-RE0>
*** FINAL System shutdown message from dhanks@R1-RE0 ***

System going down IMMEDIATELY
```

This method is a bit more verbose but allows you to retain the majority of the configuration on the routing engines.

Summary

The configuration of virtual chassis is very straightforward and should seem very familiar if you have already used virtual chassis on the Juniper EX series. The configuration of the first member in the virtual chassis is the most critical. There is a laundry list of items that need to be configured before the second member is added to the virtual chassis. The routing engine apply-groups need to be updated, the virtual chassis configuration needs to be created based off the chassis serial numbers of each member, and GRES and NSR need to be enabled. Once the foundation has been built, adding members into the virtual chassis becomes a plug and play exercise.

This chapter focuses on how virtual chassis is implemented on the Juniper MX and keeps the content highly technical and assumes you already know the basics. For more information about virtual chassis, please check out *Junos Enterprise Switching*.

VCP Interface Class of Service

Depending on the traffic patterns in the network, it's possible to cause congestion on the VCP interfaces. Recall that the VCP interfaces should be sized to roughly 50% of the total aggregate transit traffic flowing through the member. Even with properly sized VCP interfaces, it's just a fact of life that there are microbursts of traffic that will cause an interface to become congested. It's important to remember that in addition to inter-member transit traffic, the VCP interfaces also transmit the VCCP control traffic. If and when the VCP interfaces become congested due to microbursts of traffic, there needs to be a guarantee in place that gives control traffic priority so that the virtual chassis isn't negatively impacted.

VCP Traffic Encapsulation

All traffic that's transmitted across the VCP interfaces is encapsulated in an IEEE 802.1Q header that allows VCP to set the proper IEEE 802.1p code points for traffic differentiation. There are various types of traffic that use the VCP interfaces, as shown in Table 6-5.

Table 6-5. VCP Interface Traffic to IEEE 802.1p Mapping.

Traffic	Forwarding Class	Packet Loss Priority	IEEE 802.1p Code Point
PFE ↔ PFE	`best-effort`	Low	000
PFE ↔ PFE	`best-effort`	High	001
PFE ↔ PFE	`assured-forwarding`	Low	010
PFE ↔ PFE	`assured-forwarding`	High	011
PFE ↔ PFE	`expedited-forwarding`	Low	100
PFE ↔ PFE	`expedited-forwarding`	High	101
RE ↔ RE and RE ↔ PFE	`network-control`	Low	110
VCCP	`network-control`	High	111

As transit traffic flows across VCP interfaces, there could be IEEE 802.1p or DSCP code points that need to be honored. By default, Junos reserves 95% for best effort and 5% for network control. The default configuration poses two challenges: 5% of the VCP interface bandwidth isn't enough for a large virtual chassis in addition to regular control traffic, and the default configuration doesn't honor expedited and assured forwarding.

VCP Class of Service Walkthrough

VCP interfaces are able to work directly with the Junos class of service configuration without any special requirements. From the perspective of the Junos class of service daemon (`cosd`), the VCP is just another interface. Let's take a look at a life of a transit packet in a virtual chassis and how class of service is applied at each stage as it moves from the ingress interface, through the virtual chassis, and finally to the egress interface.

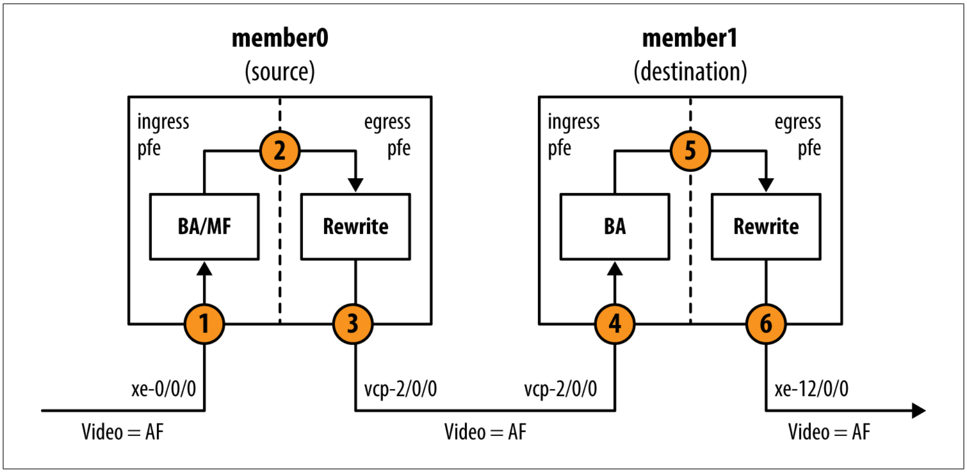


Figure 6-20. Illustration of VCP Class of Service Walkthrough.

Let's assume that a video server is connected to port xe-0/0/0, as illustrated in Figure 6-20; the egress port xe-12/0/0 is on another member in the virtual chassis and needs to traverse the VCP interface vcp-2/0/0 to each member1. Let's also assume that both interfaces xe-0/0/0.0 and xe-12/0/0.0 are `family inet` and use DSCP for classification.

1. The packet is subject to classification as it enters the xe-0/0/0 interface on member0; the classification can be in the form of a behavior aggregate or multifield classification. The end result is that the packet needs to be classified into a forwarding class. For this example, let's assume the video packets are classified into the `assured-forwarding` forwarding class.
2. The queue information is carried throughout the switch fabric in a special fabric tag. The forwarding class configuration determines the switch fabric priority. In this example, the `best-effort` and `assured-forwarding` have a switch fabric priority of low whereas `expedited-forwarding` and `network-control` have a switch fabric priority of high.
3. The video packet is queued on interface vcp-2/0/0 on member0 in the scheduler associated with the `assured-forwarding` forwarding class. The vcp-2/0/0 interface on member0 has a rewrite rule for IEEE 802.1p that will give any packets in the `assured-forwarding` forwarding class a code point of 010 or 011 depending on the packet loss priority.
4. The video packet enters the vcp-2/0/0 interface on member1 and is subject to the behavior aggregate classification on port vcp-2/0/0. The packet is sent to the `assured-forwarding` forwarding class.
5. The queue information is carried throughout the switch fabric in a special fabric tag. The forwarding class configuration determines the switch fabric priority.
6. The video packet is queued on interface xe-12/0/0 on member1 with the original packet's DSCP code points as retained by the fabric header.

As you can see, there's nothing special in the class of service functions with virtual chassis. The only caveat to be aware of is that if transit data must be sent to an egress interface on another member in the virtual chassis, the VCP interfaces are subject to IEEE 802.1p classification. This means that you need to create a consistent DSCP to IEEE 802.1p rewrite rule that can be applied on a per-hop behavior (PHB) on each member in the virtual chassis.

Forwarding Classes

Let's begin creating a class of service configuration with four standardized forwarding classes; this will keep the configuration simple and easy to troubleshoot. The majority of network operators will find four forwarding classes more than adequate, as shown in Table 6-6.

Table 6-6. Recommended Forwarding Classes for Virtual Chassis.

Forwarding Class	Queue Number	Switch Fabric Priority
best-effort	0	Low
assured-forwarding	1	Low
expedited-forwarding	2	High
network-control	3	High

Notice that the four forwarding classes have a 1:2 ratio with the number IEEE 802.1p code points; this will give each forwarding class both a high and low loss priority. The forwarding class configuration will be as follows:

```
class-of-service {
    forwarding-classes {
        queue 0 best-effort priority low;
        queue 1 assured-forwarding priority low;
        queue 2 expedited-forwarding priority high;
        queue 3 network-control priority high;
    }
}
```

As described in Chapter 5, the switch fabric has two queues: low and high. Looking at the four forwarding classes created for virtual chassis, it would make sense to place the two most important forwarding classes in the switch fabric high-priority queue and the two remaining forwarding classes in the switch fabric low-priority queue. This will ensure that the `expedited-forwarding` and `network-control` forwarding classes receive preferential treatment as the traffic is sprayed across the switch fabric to the egress PFE.

Schedulers

The next logical step is to create schedulers and assign them to each of the forwarding classes. The default 5% bandwidth scheduler for network control traffic isn't quite big enough to handle a large virtual chassis in addition to the routing engine-to-routing engine and routing engine-to-PFE traffic. Table 6-7 illustrates four new schedulers and the recommended settings.

Table 6-7. Recommended Schedulers for Virtual Chassis.

Scheduler Name	Forwarding Class	Transmit Rate	Buffer Size	Priority	Excess Priority	Excess Rate
s-medium-priority	network-control	10%	20%	high	high	N/A
s-high-priority	expedited-forwarding	50% + rate limit	25ms	strict-high	high	N/A

Scheduler Name	Forwarding Class	Transmit Rate	Buffer Size	Priority	Excess Priority	Excess Rate
`s-low-30`	`assured-forwarding`	30%	30%	N/A	N/A	99%
`s-low-weight`	`best-effort`	10%	10%	N/A	N/A	1%

The `network-control` forwarding class can now use up to 10% of the VCP interface's bandwidth, doubling the transmit rate from the Junos defaults. The buffer size of `network-control` has been doubled to allow for deeper queuing during congestion. The `expedited-forwarding` forwarding class is given a transmit rate of 50% and is rate limited so that it could never exceed this value; this should be more than enough to guarantee the delivery of latency sensitive packets through the virtual chassis. Also note the temporal buffer of 25 ms; this will guarantee the speedy delivery of latency-sensitive traffic. The `assured-forwarding` will receive a transmit rate of 30% whereas the `best-effort` only receives 10%. Any excess bandwidth that's left over from the `expedited-forwarding` and `network-control` forwarding class schedulers will be given to the `assured-forwarding` and `best-effort` forwarding classes, with the exception that the `assured-forwarding` forwarding class shall receive 99% of the excess bandwidth. This may sound harsh, but keep in mind that the schedulers will only enforce the transmit rates during congestion of the interface, and during the congestion certain traffic has to be guaranteed to be transmitted. In order to guarantee the transmission of a certain type of traffic requires that another type of traffic be penalized.

Let's take a look at the scheduler configuration that's derived from Table 6-7:

```
class-of-service {
    schedulers {
        s-high-priority {
            transmit-rate percent 10;
            buffer-size percent 20;
            priority high;
        }
        s-strict-high-priority {
            transmit-rate {
                percent 50;
                rate-limit;
            }
            buffer-size temporal 25k;
            priority strict-high;
        }
        s-low-30 {
            transmit-rate percent 30;
            excess-rate percent 99;
            drop-profile-map loss-priority low protocol any drop-profile low-plp;
            drop-profile-map loss-priority high protocol any drop-profile high-plp;
        }
        s-low-10 {
            transmit-rate percent 10;
            excess-rate percent 1;
```

```
                drop-profile-map loss-priority low protocol any drop-profile low-plp;
                drop-profile-map loss-priority high protocol any drop-profile high-plp;
            }
        }
    }
```

The final step is to create a scheduler map that will assign a specific scheduler to a particular forwarding class. Using the information in Table 6-7, the following scheduler map is created:

```
    class-of-service {
        scheduler-maps {
            sm-vcp-ifd {
                forwarding-class network-control scheduler s-medium-priority;
                forwarding-class expedited-forwarding scheduler s-high-priority;
                forwarding-class assured-forwarding scheduler s-high-weight;
                forwarding-class best-effort scheduler s-low-weight;
            }
        }
    }
```

The next step is to apply the scheduler map `sm-vcp-ifd` to all of the VCP interfaces within the virtual chassis:

```
    class-of-service {
        traffic-control-profiles {
            tcp-vcp-ifd {
                scheduler-map sm-vcp-ifd;
            }
        }
        interfaces {
            vcp-* {
                output-traffic-control-profile tcp-vcp-ifd;
            }
        }
    }
```

The use of an output traffic control profile is required to enforce schedulers that use remainder and excess calculations. A traffic control profile called `tcp-vcp-ifd` was created and references the scheduler map `sm-vcp-ifd`, which maps the various schedulers to the correct forwarding class. Each VCP interface is then assigned an output traffic control profile of `tcp-vcp-ifd`.

Classifiers

The next step is to create a behavior aggregate classifier that is to be applied to all VCP interfaces. As traffic is received on a VCP interface, the behavior aggregate will inspect the IEEE 802.1p code point and place the packet in the appropriate forwarding class, as illustrated in Table 6-8.

Table 6-8. Recommended Behavior Aggregate for Virtual Chassis.

IEEE 802.1p Code Point	Packet Loss Priority	Forwarding Class
000	Low	best-effort
001	High	best-effort
010	Low	assured-forwarding
011	High	assured-forwarding
100	Low	expedited-forwarding
101	High	expedited-forwarding
110	Low	network-control
111	High	network-control

Let's review the classification configuration based off Table 6-8:

```
class-of-service {
    classifiers {
        ieee-802.1 vcp-classifier {
            forwarding-class best-effort {
                loss-priority low code-points 000;
                loss-priority high code-points 001;
            }
            forwarding-class assured-forwarding {
                loss-priority low code-points 010;
                loss-priority high code-points 011;
            }
            forwarding-class expedited-forwarding {
                loss-priority low code-points 100;
                loss-priority high code-points 101;
            }
            forwarding-class network-control {
                loss-priority low code-points 110;
                loss-priority high code-points 111;
            }
        }
    }
}
```

The IEEE 802.1p classifier **`vcp-classifier`** has been configured using the information listed in Table 6-8. The next step is to apply the behavior aggregate to all VCP interfaces in the virtual chassis:

```
class-of-service {
    interfaces {
        vcp-* {
            unit * {
                classifiers {
                    ieee-802.1 vcp-classifier;
                }
            }
        }
        }
    }
```

The behavior aggregate has successfully been applied to all VCP interfaces within the virtual chassis. It's important to create a consistent classification and rewrite rule so that as a packet travels through a set of routers the PHB remains the same and the preferential treatment of the packet is guaranteed end to end.

Rewrite Rules

The final step is to create a rewrite rule that's consistent with the behavior aggregate. As traffic is transmitted on a VCP interface, it's critical that the forwarding classes have the appropriate IEEE 802.1p code points to enforce the end-to-end preferential treatment of packets across the virtual chassis, as shown in Table 6-9.

Table 6-9. Recommended Rewrite Rule for Virtual Chassis.

Forwarding Class	Packet Loss Priority	Code Point
best-effort	Low	000
best-effort	High	001
assured-forwarding	Low	010
assured-forwarding	High	011
expedited-forwarding	Low	100
expedited-forwarding	High	101
network-control	Low	110
network-control	High	111

Let's review the recommended rewrite policy based off the information in Table 6-9:

```
    class-of-service {
        rewrite-rules {
            ieee-802.1 vcp-rules {
                forwarding-class best-effort {
                    loss-priority low code-point 000;
                    loss-priority high code-point 001;
                }
                forwarding-class assured-forwarding {
                    loss-priority low code-point 010;
                    loss-priority high code-point 011;
                }
                forwarding-class expedited-forwarding {
                    loss-priority low code-point 100;
                    loss-priority high code-point 101;
                }
                forwarding-class network-control {
                    loss-priority low code-point 110;
                    loss-priority high code-point 111;
                }
            }
        }
    }
```

The next step is to apply the rewrite rule to all VCP interfaces:

```
    class-of-service {
        interfaces {
            vcp-* {
                unit * {
                    rewrite-rules {
                        ieee-802.1 vcp-rules;
                    }
                }
            }
        }
    }
```

Final Configuration

Each of the major class of service components has been carefully constructed and designed to give preferential treatment to control plane traffic and any user traffic placed into the `expedited-forwarding` forwarding class. All other traffic is given any remainder and excess bandwidth during times of congestion.

Let's put all of the pieces together into a final recommended configuration for virtual chassis VCP interfaces:

```
    class-of-service {
        classifiers {
            ieee-802.1 vcp-classifier {
                forwarding-class best-effort {
                    loss-priority low code-points 000;
                    loss-priority high code-points 001;
                }
                forwarding-class assured-forwarding {
                    loss-priority low code-points 010;
                    loss-priority high code-points 011;
                }
                forwarding-class expedited-forwarding {
                    loss-priority low code-points 100;
                    loss-priority high code-points 101;
                }
                forwarding-class network-control {
                    loss-priority low code-points 110;
                    loss-priority high code-points 111;
                }
            }
        }
        drop-profiles {
            low-plp {
                fill-level 70 drop-probability 1;
            }
            high-plp {
```

```
            interpolate {
                fill-level [ 25 50 75 ];
                drop-probability [ 50 75 90 ];
            }
        }
    }
    forwarding-classes {
        queue 0 best-effort priority low;
        queue 1 assured-forwarding priority low;
        queue 2 expedited-forwarding priority high;
        queue 3 network-control priority high;
    }
    traffic-control-profiles {
        tcp-vcp-ifd {
            scheduler-map sm-vcp-ifd;
        }
    }
    interfaces {
        vcp-* {
            output-traffic-control-profile tcp-vcp-ifd;
            unit * {
                classifiers {
                    ieee-802.1 vcp-classifier;
                }
                rewrite-rules {
                    ieee-802.1 vcp-rules;
                }
            }
        }
    }
    rewrite-rules {
        ieee-802.1 vcp-rules {
            forwarding-class best-effort {
                loss-priority low code-point 000;
                loss-priority high code-point 001;
            }
            forwarding-class assured-forwarding {
                loss-priority low code-point 010;
                loss-priority high code-point 011;
            }
            forwarding-class expedited-forwarding {
                loss-priority low code-point 100;
                loss-priority high code-point 101;
            }
            forwarding-class network-control {
                loss-priority low code-point 110;
                loss-priority high code-point 111;
            }
        }
    }
    scheduler-maps {
        sm-vcp-ifd {
            forwarding-class network-control scheduler s-high-priority;
            forwarding-class expedited-forwarding scheduler s-strict-high-priority;
            forwarding-class assured-forwarding scheduler s-low-30;
            forwarding-class best-effort scheduler s-low-10;
        }
    }
    schedulers {
        s-high-priority {
            transmit-rate percent 10;
            buffer-size percent 20;
            priority high;
        }
        s-strict-high-priority {
            transmit-rate {
                percent 50;
                rate-limit;
            }
            buffer-size temporal 25k;
            priority strict-high;
        }
        s-low-30 {
            transmit-rate percent 30;
            excess-rate percent 99;
            drop-profile-map loss-priority low protocol any drop-profile low-plp;
            drop-profile-map loss-priority high protocol any drop-profile high-plp;
        }
        s-low-10 {
            transmit-rate percent 10;
            excess-rate percent 1;
            drop-profile-map loss-priority low protocol any drop-profile low-plp;
            drop-profile-map loss-priority high protocol any drop-profile high-plp;
        }
    }
}
```

Verification

Once the recommended class of service configuration has been committed, it is best practice to verify the results with a few show commands. A good place to start is reviewing the forwarding classes:

```
{master:member0-re0}
dhanks@R1-RE0>show class-of-service forwarding-class
Forwarding class         ID  Queue  Restricted  Fabric     Policing   SPU
                                     queue      priority   priority   priority
  best-effort            0   0          0        low        normal     low
  assured-forwarding     1   1          1        low        normal     low
  expedited-forwarding   2   2          2        high       normal     low
  network-control        3   3          3        high       normal     low
```

The four forwarding classes have successfully installed and are showing the correct queue number and switch fabric priority.

Let’s confirm the classification and rewrite rules for the VCP interface vcp-2/0/0 on member0:

```
1    {master:member0-re0}
2    dhanks@R1-RE0> show class-of-service interface vcp-2/0/0
3    Physical interface: vcp-2/0/0, Index: 128
4    Queues supported: 8, Queues in use: 4
5      Output traffic control profile: tcp-vcp-ifd, Index: 31002
6      Congestion-notification: Disabled
7
8      Logical interface: vcp-2/0/0.32768, Index: 64
9        Object                  Name               Type               Index
10        Rewrite                 vcp-rules          ieee8021p (outer)     34
11        Classifier              vcp-classifier     ieee8021p             11
```

Line 5 confirms that the correct traffic control profile has been applied to the VCP interface. Lines 10 and 11 also confirm that the correct classifier and rewrite rule has been applied.

Let's take a look at the `show interfaces` command and confirm that the proper forwarding classes and schedulers have been installed:

```
{master:member0-re0}
dhanks@R1-RE0> show interfaces xe-2/0/0 extensive | find "CoS information"
  CoS information:
    Direction : Output
    CoS transmit queue                Bandwidth      Buffer    Priority   Limit
                           %                bps   %    usec
    0 best-effort          0                  0   r       0         low    none
    1 assured-forwarding   0                  0   r       0         low    none
    2 expedited-forwarding 90        9000000000   r       0        high    none
     3 network-control      10        1000000000   r       0 medium-high    none
```

All four forwarding classes are correct and show the proper bandwidth and priority assignments.

Summary

Virtual chassis is a powerful tool in the hands of a network engineer. Being able to provide standard high-availability features such as GRES and NSR that span different routing engines in different chassis is a simple but elegant method to mitigate risk when providing chassis virtualization. Besides the obvious OSS/BSS benefits of virtual chassis, the most often overlooked and powerful feature comes from the ability to add and remove members from a virtual chassis; this creates a "plug and play" environment where a new member can be installed into a virtual chassis to immediately scale the number of ports providing network services.

Both Enterprise and Service Provider customers can instantly benefit from virtual chassis. The administration benefits of managing and operating a single control plane versus an entire army of routers have obvious and immediate impacts. Being able to present a single system to SNMP collectors, syslog hosts, and AAA services makes everyone's life easier. Virtual chassis grants the network operator a single control plane and command line from which to make changes to the system, thereby removing the nuisance of wondering which router to log into to change a particular function.

With great power comes great responsibility. The only downside to virtual chassis is that it makes it much easier to propagate a mistake. For example, if you were modifying a routing protocol setting and made a mistake, it would impact the entire virtual chassis. Virtual chassis is subject to fate sharing; there's no way of getting around it. One method of helping ensure that critical components of the configuration aren't changed by mistake is to deploy Junos automation. There is a feature in Junos automation called commit scripts. These scripts are executed each time the configuration is committed. The scripts can be programmed to check certain values in the configuration and ensure critical components are not removed or do not exceed certain thresholds. A good example could be that any interface that contains the word "CORE" in the description must have an MTU of 4000 or the commit will fail. To learn more about Junos automation and commit scripts check out *This Week: Applying Junos Automation* (*http://www.juniper.net/us/en/community/junos/training-certification/day-one/automation-series/applying-junos-automation/*) by Juniper Networks.

Chapter Review Questions

1. Can Virtual Chassis be used with DPC line cards?
   a. Yes
   b. No

2. Which routing engine will the `ksyncd` process be running on the VC-B?
   a. Master routing engine
   b. Backup routing engine
   c. Both routing engines
   d. None of the above
3. Can you login to routing engines in the virtual chassis through their respective `fxp0` interface?
   a. Yes
   b. No
4. Which unique identifier is used when configuring a preprovisioned virtual chassis?
   a. System MAC address
   b. Chassis serial number
   c. Backplane serial number
   d. Manually assigned
5. Assuming three Juniper MX960s were in a virtual chassis, what would be the FPC number of a line card installed into slot 7 on member 2?

    a. 26
    b. 27
    c. 33
    d. 34

6. Would there be a mastership election if the VC-Mb routing engine failed?
    a. Yes
    b. No

7. What's the new apply-group naming format for routing engines in a virtual chassis?
    a. member0-re0
    b. VC-Mm
    c. vc-mm
    d. Member0-re0

8. How does the VCCP implementation on the Juniper MX build the SPF tree?
    a. Per Trio chipset
    b. Per chassis
    c. Per system MAC address
    d. Per loopback address

Chapter Review Answers

1. **Answer: B**. Virtual Chassis on the Juniper MX can only be used with Trio-based MPC line cards.

2. **Answer: C**. The VC-B member of the virtual chassis has the privilege of running the kernel synchronization process on both routing engines. Recall that the VC-B will run `ksyncd` on the VC-Bm so that it can be synchronized with the VC-Mm. The VC-Bb will also need to run another copy of `ksyncd` so that it can keep synchronized with the VC-Bm in case there's a failure.

3. **Answer: B**. Only the VC-Mm routing engine will respond on its `fxp0` interface while all other routing engines will not. You must use the console or the `request routing-engine login` command to login to other routing engines.

4. **Answer: B**. The chassis serial number is used when configuring a preprovisioned virtual chassis. No other serial number is valid.

5. **Answer: C**. Recall that the global FPC number = (member-id * 12) + local FPC. In this case, the answer would be (2 * 12) + 7 = 33.

6. **Answer: B**. There's no mastership election process when the VC-Mb fails. The only two routing engines that would cause a mastership election process in the event of a failure are the VC-Mm and VC-Bm.

7. **Answer: A**. The new apply-group naming convention for routing engines in a virtual chassis is member0-re0, member0-re0, member1-re0, member1-re1, so on and so forth.
8. **Answer: B**,**C**. Trick question. The Juniper MX VCCP implementation builds the SPF tree per chassis and uses the system MAC address as the identifier.

CHAPTER 7

Trio Inline Services

This chapter will cover Trio Inline Services and enumerate the different services that are available through the power of Trio. Many Juniper MX customers often ask, "Why is the bulk of the cost in the line cards?" The answer is because all of the *awesome* is in the line cards! Think about all of the typical services in the line cards from Chapter 1: line-rate forwarding, class of service, access lists, and much more. That's just the tip of the iceberg. Trio inline services go above and beyond and introduce sampling, network address translation, port mirroring, and tunnel protocols.

What are Trio Inline Services?

Providing additional networking services on top of routing and switching is critical to the success of any good network architecture. Networking services includes features such as:

- Sampling information and statistics
- Network Address Translation (NAT)
- Port mirroring
- Generic Routing Encapsulation (GRE) and IP tunneling
- Logical tunnels

Historically, routers have required a separate Services Module to provide additional features. One of the key differentiators of the Trio chipset is its ability to integrate network services without the requirement of an additional Services Module. With each new Junos release, it's possible to add new features such as inline services within the Trio chipset.

Providing network services as part of the MPC line cards, which use the Trio chipset, offers some distinct advantages:

Total Cost of Ownership

As network services become available through the Trio chipset, you are no longer required to purchase an additional Services Module. However, the real cost savings manifests itself in an additional FPC slot that is now available because the Services Module isn't required. This additional FPC can be used to provide additional WAN ports to realize previously lost revenue.

Configuration

The configuration of inline Trio services is largely the same as services on the MS-DPC. The largest difference is the requirement to set aside a specific amount of bandwidth on the Trio chipset to be reserved for inline services.

Performance

Inline Trio services are processed directly on the Lookup Block, which enables near line-rate performance of network services. The Trio chipset is responsible for forwarding traffic; by being able to apply network services as part of the same workflow, it's possible to provide near line-rate performance as opposed to having to send the packet to a separate Services Module.

The biggest advantage is that enabling Trio inline services doesn't require the loss of any physical WAN ports. Configuring Trio inline services requires that a certain amount of bandwidth to be specified for processing network services; this bandwidth will be subtracted from the available WAN ports during congestion. For example, if you were to configure 10 Gbps of bandwidth for Trio inline services on an MPC with 40 Gbps of WAN ports, during congestion only 30 Gbps would be available to the WAN ports while 10 Gbps is available for Trio inline services. In summary, the WAN ports and Trio inline services will share the available chipset bandwidth.

J-Flow

Perhaps one of the most popular network services is J-Flow. J-Flow allows you to sample a subset of traffic and collect flow statistics. Flows are identified by unidirectional conversations. A flow is uniquely identified by the following fields:

- Source IP address
- Destination IP address
- Source port number
- Destination port number
- Protocol
- Type of service
- Ingress interface

Collecting flow information is critical for businesses to provide accounting and billing, network capacity planning, and traffic profiling and analysis, and some countries require by law that all connections be collected. Flow information is created and transmitted to an external collector for further processing.

J-Flow Evolution

J-Flow is used to describe many different variants of collecting flow statistics. Each successive version of J-Flow provides more features and functionality than the previous version.

J-Flow v5
: This version of J-Flow supports only IPv4 and fixed fields that are not user-configurable.

J-Flow v8
: Flow aggregation was added with J-Flow v8. This enables the router to use less bandwidth, sending flow statistics to collectors. Another benefit is that the aggregation reduces the memory requirements of the collectors.

J-Flow v9
: The introduction of RFC 3954 introduced new concepts into flow statistics. The most notable was the introduction of predefined templates such as IPv4, IPv6, MPLS, and IPv4 in MPLS. Templates allow the router and collector to describe the flow fields in a common language. This allows the protocol to be scaled to support new applications with a new template to describe the flow fields.

IP Flow Export Information/J-Flow v10
: The latest version of flow statistics is IP Flow Export Information (IPFIX), which is based on RFC 5101, 5102, and 5103. IPFIX is the official IETF protocol that was created based on the need for a common and universal standard of exporting flow information. There's little change between IPFIX and J-Flow v9 aside from some cosmetic message headers and the introduction of variable-length fields.

Inline IPFIX Performance

Because inline IPFIX is implemented within the Trio chipset, the performance is near line rate. As traffic moves through the Trio chipset, it's able to be inspected and sampled locally without having to take a longer path to a Service Module and back. Being able to keep the packet within the Trio chipset speeds up the operations and lowers the latency.

Table 7-1. Inline IPFIX Performance Chart.

What	MPC1	MPC2	MPC3E
Max flow records	4 M	8 M	16 M
Flow setup rate	150 K flows/second	300 K flows/second	600 K flows/second
Flow export rate	100K flows/second	200 K flows/second	400 K flows/second

What	MPC1	MPC2	MPC3E
Throughput	20 Gbps	40 Gbps	80 Gbps
Maximum Packets Per Second	15 Mpps	30 Mpps	60 Mpps
Trio Lookup Blocks	1	2	4

The performance of inline IPFIX is directly related to how many Lookup Blocks are available for processing. In the MPC1, there's only a single Trio chipset available; the MPC2 has two Trio chipsets, so effectively doubles the performance. The most interesting is the MPC3E line card; it has a single Trio chipset, but within the chipset has four Lookup Blocks. Because the MPC3E has four Lookup Blocks, it effectively has four times the performance of the MPC1.

The performance numbers listed in Table 7-1 are current as of Junos 11.4 on current generation hardware. These numbers are subject to change with new code releases and hardware. Please consult *http://www.juniper.net/* or your account team for more accurate numbers.

Inline IPFIX Configuration

The configuration of inline IPFIX has four major parts: chassis FPC, flow monitoring, sampling instance, and firewall filters. Each component is responsible for a unique set of items, that when combined produce a working model to sample traffic.

Each component is used like building blocks to build an inline IPFIX. Figure 7-1 illustrates that FPCs within a chassis are associated with a particular sampling instance. Each sampling instance is associated with an IPFIX template. Multiple FPCs can share the same sampling instance, and multiple sampling instances can share the same template. The only restriction is that a FPC can only be associated with one sampling instance.

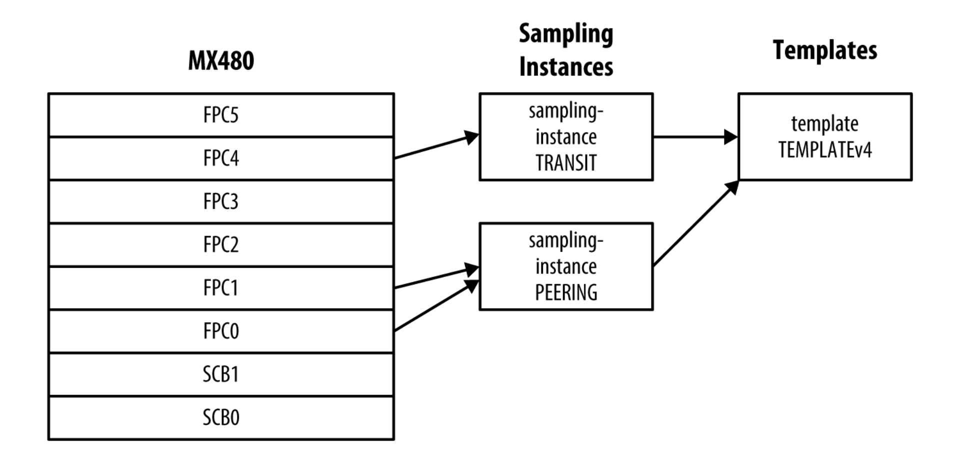


Figure 7-1. Inline IPFIX Configuration Components.

Chassis Configuration

The first step in configuring inline IPFIX is to define which FPC requires sampling. Inline IPFIX is implemented on a per-FPC basis and must follow a couple rules:

- One FPC can support only a single instance.
- Multiple families can be configured per instance.

Let's review a sample inline IPFIX chassis configuration as illustrated in Figure 7-1:

```
chassis {
    fpc 0 {
        sampling-instance PEERING;
    }
    fpc 1 {
        sampling-instance PEERING;
    }
    fpc 4 {
        sampling-instance TRANSIT;
    }
}
```

This effectively binds the `sampling-instance PEERING` with FPC0 and FPC1 and `sampling-instance TRANSIT` FPC4. The instance and template information will be assigned and downloaded to the Lookup Block on each respective FPC. Now that the FPC to `sampling-instance` association has been made, the next step is to configure the flow monitoring.

Flow Monitoring

Because IPFIX was based on J-Flow v9, templates are required because they must be associated with a collector. To define IPFIX templates, the configuration will be placed into the [services flow-monitoring version-ipfix] stanza. Let's review an example template called TEMPLATEv4 that has been configured for IPv4:

```
services {

    flow-monitoring {

        version-ipfix {

            template TEMPLATEv4 {
                flow-active-timeout 150;
                flow-inactive-timeout 100;
                template-refresh-rate {
                    seconds 10;
                }

                option-refresh-rate {
                    seconds 10;
                }

                ipv4-template;
            }
        }
    }
}
```

The creation of templates allows for customized settings that can be associated with different collectors. There are four major settings that are available when creating a template:

Active Flow Timeout
: This option is adjusted with the flow-active-timeout knob. Use this setting to specify the number of seconds between export updates. This can be useful to break up the reporting of long lived sessions.

Inactive Flow Timeout
: This option is adjusted with the flow-inactive-timeout knob. This option is used to determine when a particular flow is considered inactive and can be purged from the flow table. For example, if flow-inactive-timeout is configured for 100, the router will wait for 100 seconds of inactivity on a particular flow; once a flow has been inactive for 100 seconds, the router will send a final flow export to the collector and purge the flow.

Template Refresh Rate and Option Refresh Rate
: Every so often, the template data needs to be refreshed and transmitted to the collector. The template-refresh-rate and option-refresh-rate knobs give you choices for setting the frequency: seconds or packets. By using the seconds keyword, the router will transmit the template data to the collector at the specified interval. The packets keyword will transmit the template every *N* packets. Setting both template-refresh-rate and option-refresh-rate is required when adjusting the template update frequency.

Template

The final piece of information is what type of template should be used and sent to the collector. As of Junos 11.4, the only template type available for inline IPFIX is `ipv4-template`.

Once the flow monitoring templates have been configured, they can be used as a reference when building the sampling instance. The next step is to configure the sampling instances, which bridge together the FPC and templates.

Sampling Instance

The sampling instances are the glue that brings together all the different pieces of the inline IPFIX configuration. A sampling instance can have multiple families and each family can reference a different template, as shown in Figure 7-2.

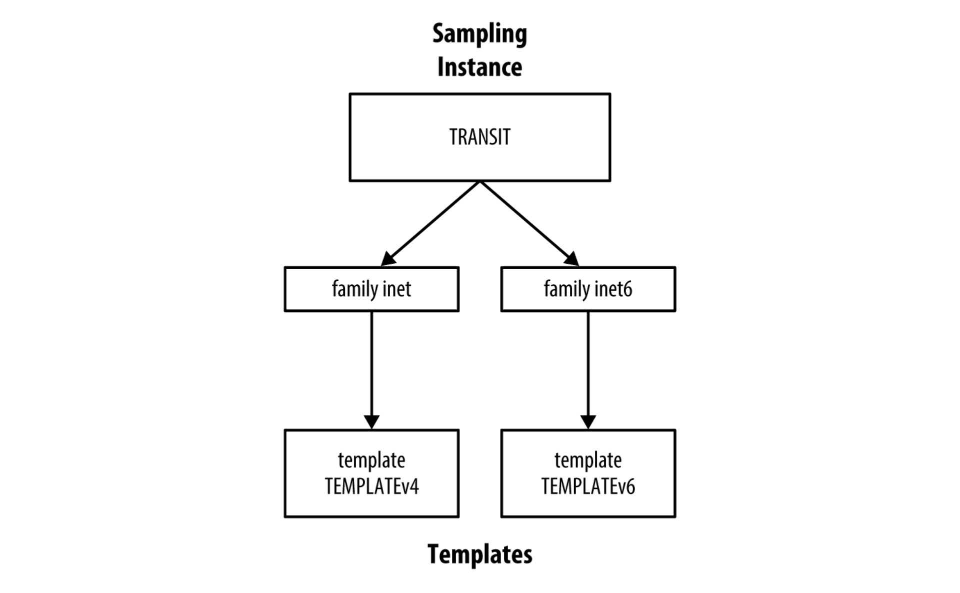


Figure 7-2. Inline IPFIX Sampling Instance Family to Template.

Although sampling instances and templates don't support IPv6 as of Junos 11.4, it's important to visualize how the sampling instance references families and how families are associated with a template.

There are many components when configuring a sampling instance. The components work together to define how flows are sampled and exported to a collector. Let's step through each component one by one:

Input Rate

This option defines the sampling rate. Other methods of sampling such as with the routing engine or MS-DPC required that the `rate` and `run-length` be defined, as shown in Figure 7-3.

$$\boxed{\text{packets\_sampled} = \frac{\text{run-length} + 1}{\text{rate}}}$$

Figure 7-3. Packets Sampled Formula for Routing Engine and MS-DPC Sampling.

When using inline IPFIX, the only valid `rate` is 1. The option `run-length` isn't configurable, because there's no need to sample data from the perspective of the microcode in the Trio Lookup Block. Every packet will be inspected and subject to flow export.

Family

Multiple families can be configured inside of a sampling instance. However, as of Junos 11.4, the only supported family is `inet`. Each family allows the configuration of collectors and associated options.

Flow Server

This option specifies a collector to which the flows will be exported. This can be in either IPv4 or IPv6 format.

Port

This option specifies which UDP port to use when exporting flows.

IPFIX Template

Templates are required when exporting flows to a collector. Recall that templates were created previously in this chapter. This option associates a template with the collector of the specified family.

Source Address

Because inline IPFIX uses UDP to export the flows, this value is considered cosmetic. Insert any source IP address that you would like to see; however, a common practice is to use the router's loopback address.

Now let's put all of the components together and build an example sampling instance that accurately reflects Figure 7-1. There needs to be two sampling instances created: `PEERING` and `TRANSIT`. Each sampling instance will be associated with the `TEMPLATEv4`.

```
forwarding-options {
    sampling {
        instance {
            PEERING {
                input {
                    rate 1;
                }
                family inet {
                    output {
                        flow-server 192.0.2.122 {
                            port 2055;
                            version-ipfix {
                                template {
                                    TEMPLATEv4;
                                }
                            }
                        }
                        inline-jflow {
                            source-address 10.7.255.3;
                        }
                    }
                }
            }
            TRANSIT {
                input {
                    rate 1;
                }
                family inet {
                    output {
                        flow-server 192.0.2.123 {
                            port 2055;
                            version-ipfix {
                                template {
                                    TEMPLATEv4;
                                }
                            }
                        }
                        inline-jflow {
                            source-address 10.7.255.3;
                        }
                    }
                }
            }
        }
    }
}
```

In this example, the only difference between the two sampling instances `PEERING` and `TRANSIT` is the **`inline-jflow source-address`**. Now that the chassis configuration, flow monitoring, and sampling instances have been configured, traffic still isn't subject to being sampled. The last remaining piece of configuration to trigger the sampling of traffic is a firewall filter.

Firewall Filter

With all of the major inline IPFIX configuration components in place, the last step is to identify which traffic needs to be sampled. If you think of the chassis configuration, flow monitoring, and sampling instances as just a big machine for inline IPFIX, it's easy to imagine feeding packets into this machine to be sampled; the beauty of Junos is that building firewall filters is an excellent tool to do just this.

Leveraging firewall filters to identify traffic opens up a lot of possibilities. Imagine being able to selectively sample customer traffic or only traffic with a certain class of service. Perhaps you require a bigger hammer and want to sample all traffic flowing through an interface. For simplicity, let's stick with the big hammer approach and sample everything:

```
firewall {
    family inet {
        filter SAMPLE-ALL {
            term 1 {
                then sample;
            }
        }
    }
}
```

The firewall filter `SAMPLE-ALL` can be used to sample all IPv4 traffic. The last step is to apply it to an interface to which you want to sample traffic from. Recall that in the example, inline IPFIX configurations that FPC0, FPC1, and FPC4 were associated with sampling instances, so only interfaces on these FPCs can be sampled. If you wish to sample traffic on an interface that lives on a different FPC, it's required to associate that FPC with a sampling instance.

```
interfaces {
    xe-0/0/0 {
        vlan-tagging;
        unit 0 {
            family bridge {
                interface-mode trunk;
                vlan-id-list 1-999;
            }
        }
        unit 1 {
            vlan-id 1000;
            family inet {
                filter {
                    input SAMPLE-ALL;
                    output SAMPLE-ALL;
                }
                address 10.8.0.0/31;
            }
            family iso;
        }
    }
}
```

In this example, any IPv4 traffic that enters or leaves the interface xe-0/0/0.1 will be subject to sampling, whereas any bridged traffic on xe-0/0/0.0 isn't sampled.

Inline IPFIX Verification

Now that inline IPFIX has been configured and traffic is being sampled, the next step is to verify that flows are being created and that sampling instances are correctly associated with FPCs.

Let's begin by checking to see if the sampling instance has been associated with the FPC:

```
1    {master}
2    dhanks@R1-RE0>  request pfe execute target fpc0 command "show sample instance
                        association"
3    SENT: Ukern command: show sample instance association
4    GOT:
5    GOT: Sampler Parameters
6    GOT: Global Sampler Association: "&global_instance"
7    GOT: FPC Bindings              :
8    GOT: sampling    : PEERING
9    GOT: port-mirroring 0 :
10    GOT: port-mirroring 1 :
11    GOT: PIC[0]Sampler Association:
12    GOT: sampling    :
13    GOT: port-mirroring 0 :
14    GOT: port-mirroring 1 :
15    GOT: PIC[1]Sampler Association:
16    GOT: sampling    :
17    GOT: port-mirroring 0 :
18    GOT: port-mirroring 1 :
19    GOT: Sampler Association
20    GOT: PFE[0]-[0]Sampler Association: "PEERING":class 1 proto 0 instance id 2
21    GOT: PFE[1]-[0]Sampler Association: "PEERING":class 1 proto 0 instance id 2
22    LOCAL: End of file
```

Lines 8 and 19 to 21 show that the sampling instance PEERING has been successfully associated with the PFE. Now let's check to see if a connection to the flow collector has been made:

```
1    {master}
2    dhanks@R1-RE0>
request pfe execute target fpc0 command "show pfe manager service_thread jflow stat"
3    SENT: Ukern command: show pfe manager service_thread jflow stat
4    GOT:
5    GOT:
6    GOT:  Sampled Connection Status          : 1
7    GOT:  Sampled Connection Retry Count     : 0
8    GOT:  Msgs received                      : 69
9    GOT:  UI Requests received               : 0
10    GOT:  Active Config                      : 1
11    GOT:  Queue Enque Retry Count            : 0
12    LOCAL: End of file
```

Line 6 indicates that the FPC has created a connection to the collector. A status of "1" indicates open, whereas a status of "0" indicates closed. Now that the FPC and connection to the collector have been verified, let's review the inline IPFIX summary:

```
{master}
dhanks@R1-RE0>
request pfe execute target fpc0 command "show services inline-jflow summary"
SENT: Ukern command: show services inline-jflow summary
GOT:
GOT:
GOT: Inline Jflow Sampling Instances:
GOT:
GOT:    Inline Instance Name              : PEERING
GOT:    Inline Instance Class             : 1
GOT:    Inline Instance Proto             : 0
GOT:
GOT:    Template Refresh Time             :10
GOT:    Option Refresh Time               :10
GOT:    Template Refresh Packets          :4800
GOT:    Option Refresh Packets            :4800
GOT:
GOT: Inline Jflow Template & Option Refresh Stats:
GOT:
GOT: Timer Expirt Counts:
GOT: ====================
GOT:    Template Refresh Timer Expiry Cnt :  52
GOT:    Option Refresh Timer Expiry Cnt   :  52
GOT:    Pkt Refresh Timer Expiry Cnt      :  4
GOT: Packet Sent Count:
GOT: ===================
GOT:    Template Refresh Sent Cnt         :  0
GOT:    Option Refresh Sent Cnt           :  0
GOT:    Template Refresh Pkt Sent Cnt     :  0
GOT:    Option Refresh Pkt Sent Cnt       :  0
GOT:
LOCAL: End of file
```

A review of the output with the configuration verifies that the values do indeed match: the template will be refreshed every 10 seconds. Let's move out of the PFE and back into the CLI. There are a few commands to let you gauge the status and flows being generated by inline IPFIX:

```
{master}
dhanks@R1-RE0> show services accounting status inline-jflow fpc-slot 0
  Status information
    FPC Slot: 0
    Export format: IP-FIX
    Route record count: 23, AS count: 2
    Route record set: Yes, Configuration set: Yes
```

This verifies the inline IPFIX configuration on FPC0. The export format is indeed IPFIX, and you can see the number of route records generated so far. Let's dig a little bit deeper and review how many flows have been found:

```
{master}
dhanks@R1-RE0> show services accounting flow inline-jflow fpc-slot 0
  Flow information
    FPC Slot: 0
    Flow packets: 214881, Flow bytes: 12708080
```

```
        Active flows: 4, Total flows: 4
        Flows exported: 4, Flows packets exported: 4
        Flows inactive timed out: 0, Flows active timed out: 4
```

Much better. You can see that nearly a quarter of a million packets have been sampled and there are a total of four active flows.

IPFIX Summary

Using inline IPFIX is an excellent method to generate basic flow statistics with the Juniper MX Series; the only drawback is that the number of families supported as of Junos 11.4 is currently limited to IPv4. However, the benefits are that it doesn't require a MS-DPC services line card and the performance and scale of inline IPFIX is much better because it runs directly in the microcode of the Trio Lookup Block.

As of Junos 11.4, there are two software licenses that enable the use of inline IPFIX:

S-ACCT-JFLOW-CHASSIS
: This license will enable J-Flow for the entire chassis. For example, an MX960 with 12 line cards would be able to use J-Flow on every line card.

S-ACCT-JFLOW-IN
: This license will enable J-Flow for a single MPC. For example, in an MX960 with 12 line cards, only a single FPC such as FPC3 can use J-Flow.

Network Address Translation

The Trio chipset supports inline Network Address Translation (NAT). The Lookup Block as of Junos 11.4 only supports simple 1:1 NAT with no port address translation. Simple NAT includes the following: source NAT, destination NAT, and two-way NAT. The primary driver for inline NAT is performance and low latency. Inline NAT is performed in the microcode of the Trio Lookup Block and doesn't require moving the packet through a dedicated Services Module.

Types of NAT

Inline Trio supports 1:1 NAT; this specifically means that IP address #1 can be translated into IP address #2. There's no port translation available, as this would require keeping track of flows and state. 1:1 NAT can be expressed in three different methods: source NAT, destination NAT, and twice NAT. In implementation, all three methods are the same; the only differences between them are the direction and number of translations.

Source NAT will inspect egress traffic from `H1` and change the source address upon translation to `H2`, as shown in Figure 7-4.

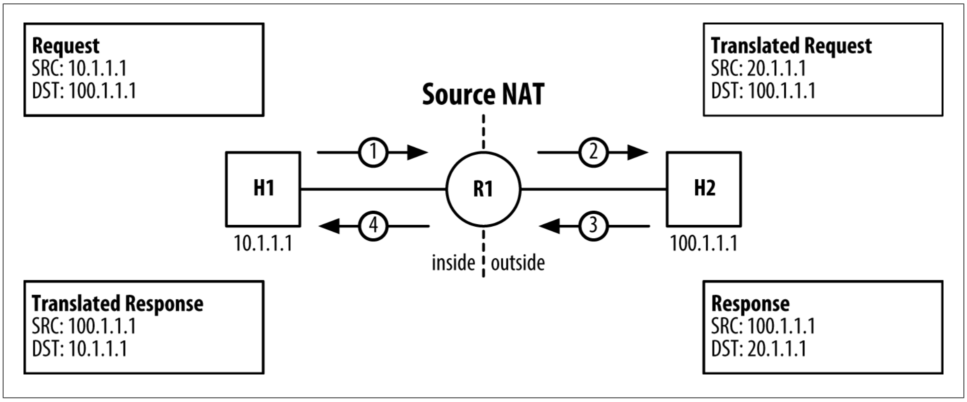


Figure 7-4. Inline Trio Source NAT.

Destination NAT will inspect egress traffic from `H2` and change the destination address upon translation to `H1`, as shown in Figure 7-5.

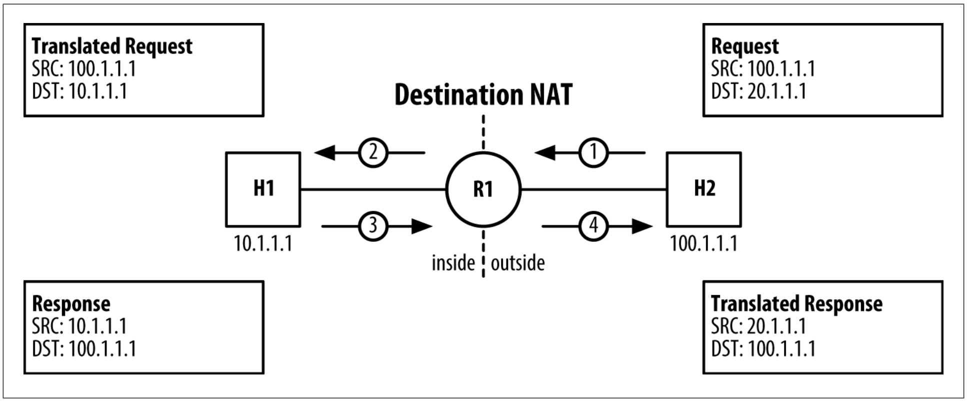


Figure 7-5. Inline Trio Destination NAT.

Twice NAT simply combines source and destination NAT together to create a scenario where both the source and destination addresses are translated. Twice NAT is helpful in use cases where the source and destination represent different customers but with the same IP address space. Egress traffic from `H1` is translated and sent to `H2` with a new source and destination address; egress traffic from `H2` is translated again and sent back to `H1` with the original source and destination IP addresses, as shown in Figure 7-6.

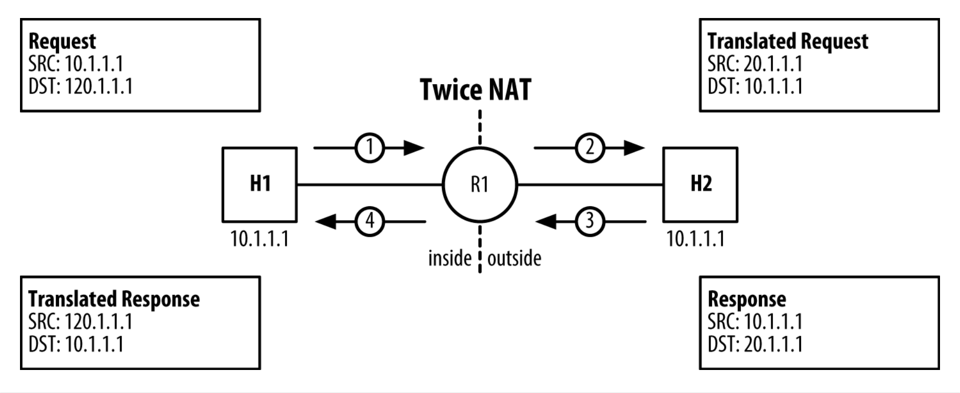


Figure 7-6. Inline Trio Twice NAT.

Even though Trio inline NAT is limited to a scale of 1:1 as of Junos 11.4, it's still able to scale and be flexible enough to satisfy several different use cases. For example, source NAT can be used to translate private customer VPN traffic that's destined to the Internet, and twice NAT can be used between two different customer VPNs that have conflicting address space.

Services Inline Interface

Prior to the Trio chipset, the only other method to implement NAT was through the MS-DPC services card. The use of the MS-DPC created a dedicated service processor logical interface called `sp-FPC/PIC/PORT`; this interface was used as an input/output device to perform services such as NAT.

With the introduction of Trio and inline NAT services, the same architecture is still in place. The only change is that inline services have introduced a new logical interface called `si-FPC/PIC/PORT`. The creation of the new `si-` interface is similar to `sp-` but requires bandwidth be set aside in a specific Trio Lookup Block.

```
chassis {
    fpc 2 {
        pic 0 {
            inline-services {
                bandwidth 1g;
            }
        }
        pic 1 {
            inline-services {
                bandwidth 1g;
            }
        }
    }
}
```

An `si-` interface can be created for each Trio Lookup Block. In this example, FPC2 is a MPC2 line card, which has two Trio chipsets. The example chassis configuration created two service inline interfaces: `si-2/0/0` and `si-2/1/0`. Let's verify:

```
dhanks@R3> show interfaces terse | match si-
si-2/0/0                up    up
si-2/1/0                up    up
```

Just as expected. The service inline interfaces followed the naming format of si-FPC/PIC/PORT. In the example, the FPC is 2, the PIC represents the Trio Lookup Block, and the port will always be 0. Let's take a closer look at the interfaces:

```
dhanks@R3> show interfaces si-2/0/0
Physical interface: si-2/0/0, Enabled, Physical link is Up
  Interface index: 145, SNMP ifIndex: 819
  Type: Adaptive-Services, Link-level type: Adaptive-Services, MTU: 9192, Speed:
1000mbps
  Device flags   : Present Running
  Interface flags: Point-To-Point SNMP-Traps Internal: 0x4000
  Link type      : Full-Duplex
  Link flags     : None
  Last flapped   : Never
  Input rate     : 1344 bps (2 pps)
  Output rate    : 0 bps (0 pps)
```

Although si-2/0/0 is a pseudo interface, it's apparent that this interface is being used for service processing by taking note of the `Type: Adaptive-Services`. With the services inline interface up and running, the next step is understand how to use this new interface.

Service Sets

The first step in configuring the router to handle services is through service sets; they define how the router applies services to packets. There are three major components required to create a service set:

Service Rules
: Similar to firewall rules, service rules match specific traffic and apply a specific action.

Type of Service Set
: Service sets have two types: next-hop style and interface style. Next-hop style relies on using the route table to forward packets into the service inline interface, and the interface style relies on defining `service-sets` directly on interfaces to forward packets into the service inline interface.

Service Interfaces
: If using the interface style approach, defining which service inline interface to use is required.

The service set implementation is located in the [services service-set] stanza of the configuration. Inline NAT supports both next-hop style and interface style configuration of service sets. Let's walk through each of the styles in detail.

Next-Hop Style Service Sets

The next-hop style service set depends on the route table to forward packets to the service inline interface. This is typically referred to as the "big hammer" approach as all traffic forwarded via the route table to the service inline interface will be subject to service rules. For example, consider the following static route:

```
routing-instances {
    CUSTOMER_A {
        instance-type vrf;
        interface si-2/0/0.1;
        interface xe-2/0/1.0;
        routing-options {
            static {
                route 10.5.0.10/32 {
                    next-hop si-2/0/0.1;
                }
            }
        }
    }
}
```

Any traffic inside of the CUSTOMER_A routing instance that's destined to the address 10.5.0.10/32 will be forwarded directly to the service inline interface si-2/0/0.1 and be subject to any service rules inside of the service set, as illustrated in Figure 7-7.

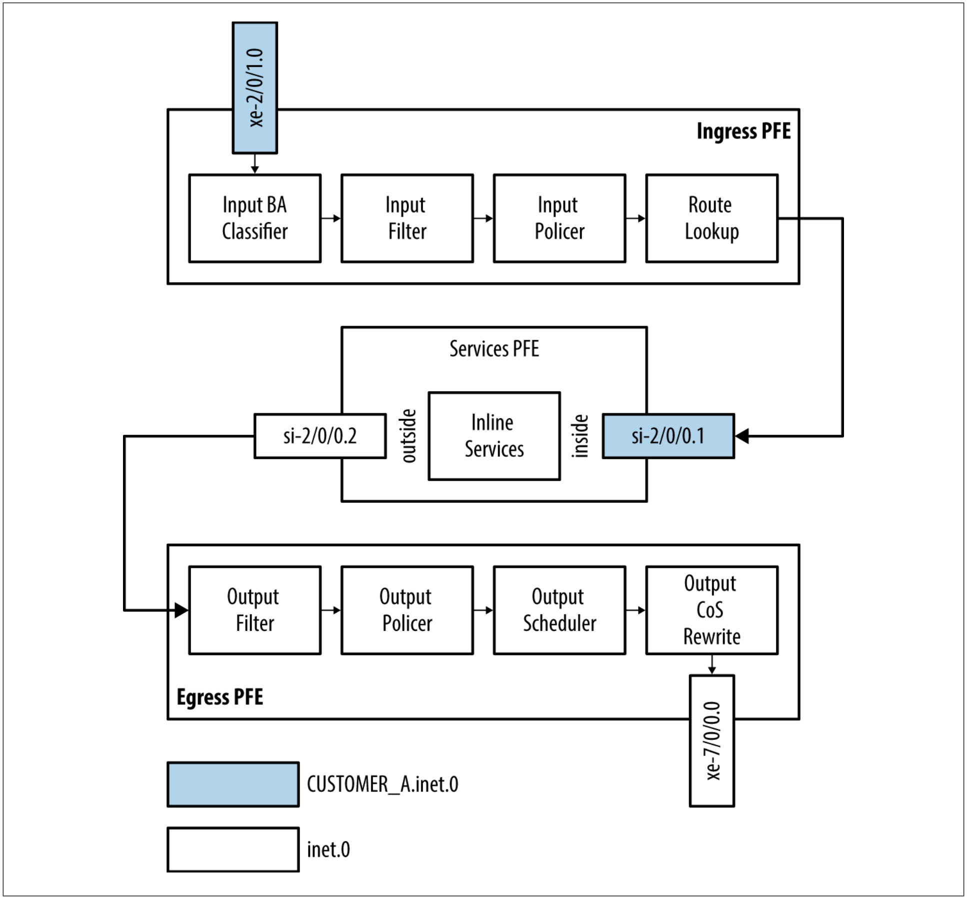


Figure 7-7. Illustration of Next-Hop Style Service Set Workflow.

Ingress traffic on xe-2/0/1.0 is part of the routing instance `CUSTOMER_A` and will be subject to the typical gauntlet of classification, filtering, policing, and route look up. Any traffic destined to 10.5.0.10/32 will be forwarded to si-2/0/0.1 as part of the `CUSTOMER_A` static route. Once the traffic enters the service inline interface, it will be subject to service rules. If the traffic matches any service rules, it will be processed. The traffic then exits the other side of the service inline interface si-2/0/0.2. The interface si-2/0/0.2 is part of the default routing instance and is subject to the typical output filter, policer, scheduling, and class of service rewriting before being transmitted as egress traffic.

Now that you have a better understanding of how traffic can be routed through the service inline interface, let's take a look at how to configure a service set using the next-hop style configuration.

```
services {
    service-set SS1 {
        nat-rules SNAT;
        next-hop-service {
            inside-service-interface si-2/0/0.1;
            outside-service-interface si-2/0/0.2;
        }
    }
}
```

The service set SS1 represents a next-hop style implementation as indicated by the nexthop-service option. The next-hop style requires explicit definition of an inside and outside service interface, as shown in Figure 7-8.

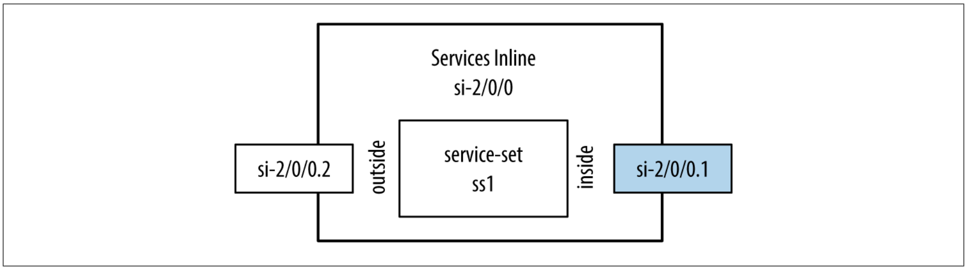


Figure 7-8. Service Set Inside and Outside Interfaces.

At this point, the service set SS1 has become an I/O machine. Traffic can enter SS1 from either si-2/0/0.1 or si-2/0/0.2. The important thing to remember is the notation of inside and outside; the service set will use inside and outside to determine the direction of traffic during the creation of service rules. For example, if the inside interface is used to route the packet, the packet direction is input, and if the outside interface is used to direct the packet to the service inline interface, the packet direction is output.

Reviewing the SS1 configuration, there's only a single service configured: nat-rules SNAT. Any traffic entering and leaving the service set will be subject to the NAT rules of SNAT:

```
    services {

        service-set SS1 {
            nat-rules SNAT;
            next-hop-service {
                inside-service-interface si-2/0/0.1;
                outside-service-interface si-2/0/0.2;
            }
        }

        nat {
            pool POOL1 {
                address 20.0.0.0/24;
            }

            rule SNAT {
                match-direction input;
                term T1 {
                    from {
                        source-address {
                            10.4.0.0/24;
                        }
                    }

                    then {
                        translated {
                            source-pool POOL1;
                            translation-type {
                                basic-nat44;
                            }
                        }
                    }
                }
            }
        }
    }
```

Now the configuration has come full circle. The service set `SS1` will use the NAT rule SNAT to determine if NAT services need to be applied. When creating a NAT rule, there are three major components:

Match Direction
The direction of traffic is expressed as either input or output. When using next-hop style services, any traffic destined to the outside interface is considered output and any traffic destined to the inside interface is considered input.

From
Just like policy statements and firewall filters, the *from* statement builds a set of conditions that must be met in order to apply an action.

Then
Once traffic has been matched with the *from* statement, the traffic is subject to any type of action and processing indicated in the *then* statement.

In the example, the NAT rule `SNAT` has a `match-direction` of `input`; because the service set is using a next-hop style implementation, this will match all traffic arriving on the inside interface si-2/0/0.1. The *from* statement only has a single match condition; any traffic that has a source address 10.4.0.0/24 will be subject to services. The *then* statement specifies that the source pool `POOL1` should be used and the traffic should be translated using `basic-nat44`.

Trio inline NAT supports the concept of NAT pools. These are pools of addresses that can be used when translating source and destination addresses. In this example, `POOL1` contains a pool of 256 addresses in the 20.0.0.0/24 network. The `basic-nat44` is an option that tells the router to use basic NAT for IPv4 traffic to IPv4 traffic. Basic NAT is defined as no network address port translation (NAPT).

This example builds a service set that will match traffic from 10.4.0.0/24 that's destined to 10.5.0.10/32 and change the source address to an address in the 20.0.0.0/24 range.

Let's review the topology of this example, as shown in Figure 7-9. S3 and S4 represent switches whereas H3 and H4 represent hosts. R3 is a Juniper MX240 with a MPC2 line card in FPC2.

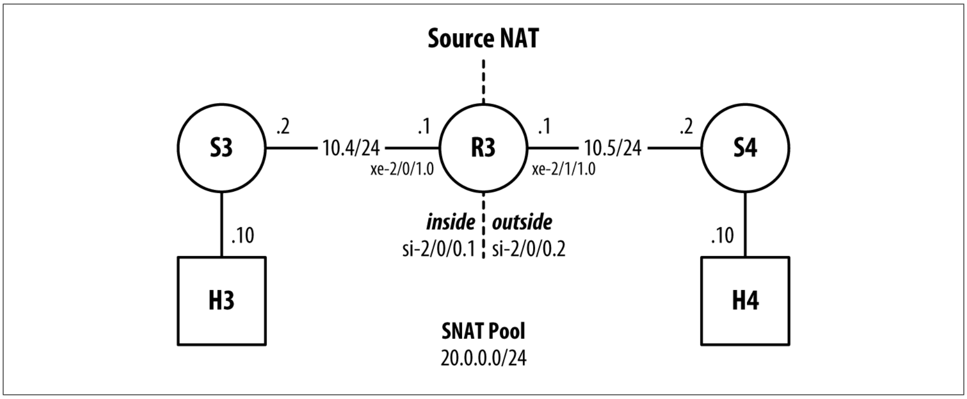


Figure 7-9. Example Trio Inline SNAT with Next-Hop Style Service Sets.

In this example, host H3 sends a ping to 10.5.0.10, and the service set SS1 on R3 matches this traffic and applies the source NAT. R3 sends the ping destined to H4, but with a new source address in the 20.0.0.0/24 pool. H4 receives the ping and replies back; the ping packet has a source address in the range of 20.0.0.0/24, and H4 has a static route of 0/0 pointing back to 10.5.0.1 via R3. S4 forwards the frame back to R3 where it matches the service set; the source NAT is undone and the response is sent back to H3.

Let's take the example to the next level and walk through each component step by step. Starting with the creation of the service inline interface:

```
chassis {
    fpc 2 {
        pic 0 {
            inline-services {
                bandwidth 1g;
            }
        }
    }
}
```

The chassis configuration will create a si-2/0/0 interface. Now let's define the inside and outside service domains and configure R3's interfaces to S3 and S4:

```
interfaces {

    si-2/0/0 {
        unit 1 {
            family inet;
            service-domain inside;
        }
        unit 2 {
            family inet;
            service-domain outside;
        }
    }

    xe-2/0/1 {
        unit 0 {
            family inet {
                address 10.4.0.1/24;
            }
        }
    }

    xe-2/1/1 {
        unit 0 {
            family inet {
                address 10.5.0.1/24;
            }
        }
    }
}
```

Just as illustrated in Figure 7-9, si-2/0/0.1 will have a service domain of inside while si-2/0/0.2 will have a service domain of outside. R3's xe-2/0/1.0 interface has an IP of 10.4.0.1/24 while xe-2/1/1.0 has an IP of 10.5.0.1/24.

The next step is to create a routing instance on R3 to contain the inside service domain and interface connected to S3. This routing instance will force traffic arriving from S3 to be placed into a separate routing table that can forward traffic to si-2/0/0.1; this interface represents the inside service domain and will expose the traffic to service sets:

```
routing-instances {
    CUSTOMER_A {
        instance-type vrf;
        interface si-2/0/0.1;
        interface xe-2/0/1.0;
        routing-options {
            static {
                route 10.5.0.10/32 {
                    next-hop si-2/0/0.1;
                }
            }
        }
    }
}
```

Now let's review the services configuration in its entirety:

```
services {
    service-set SS1 {
        nat-rules SNAT;
        next-hop-service {
            inside-service-interface si-2/0/0.1;
            outside-service-interface si-2/0/0.2;
            }
        }
```

The next-hop style service set `SS1` is created by defining the inside and outside service interfaces. `SS1` only has a single service: `nat-rules SNAT`. Let's step through the NAT configuration:

```
        nat {
            pool POOL1 {
                address 20.0.0.0/24;
            }
            rule SNAT {
                match-direction input;
                term T1 {
                    from {
                        source-address {
                            10.4.0.0/24;
                        }
                    }
                    then {
                        translated {
                            source-pool POOL1;
                            translation-type {
                                basic-nat44;
                            }
                        }
                    }
                }
            }
        }
    }
```

The NAT rule SNAT matches traffic arriving on the inside service domain (si-2/0/0.1) and will source NAT any traffic matching a source address of 10.4.0.0/24. The source NAT has a pool of addresses from the 20.0.0.0/24 network to choose from.

With the Trio inline NAT configuration in place on `R3`, let's take a look at the routing table and see how traffic will flow through the router:

```
dhanks@R3> show route

inet.0: 3 destinations, 3 routes (3 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

10.5.0.0/24        *[Direct/0] 08:25:29
                    > via xe-2/1/1.0
10.5.0.1/32        *[Local/0] 08:25:32
                      Local via xe-2/1/1.0
20.0.0.0/24        *[Static/1] 00:35:24
                    > via si-2/0/0.2

CUSTOMER_A.inet.0: 3 destinations, 3 routes (3 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

10.4.0.0/24        *[Direct/0] 00:40:24
```

```
                    > via xe-2/0/1.0
10.4.0.1/32        *[Local/0] 00:40:24
                      Local via xe-2/0/1.0
10.5.0.10/32       *[Static/5] 00:35:24
                    > via si-2/0/0.1
```

There are two routing tables, as expected: CUSTOMER_A and the default routing instance inet.0. The default route table inet.0 has the Direct route 10.5/24 on xe-2/1/1.0 as expected, and the CUSTOMER_A route table has the Direct route 10.4/24 on xe-2/0/1 and a route for 10.5.0.10/32, pushing all traffic into the service inline interface for NAT processing. However, the really interesting static route is 20.0.0.0/24 in the inet.0 route table. This route wasn't configured under routing-options, so where did it come from? The answer is that the NAT pool POOL1 automatically injected this route into the same routing instance as the output service domain interface si-2/0/0.2. Once the traffic passes through R3 and is translated with a source address from the pool 20.0.0.0/24, R3 still has to process the return traffic. The return traffic will have a destination address from the 20.0.0.0/24 pool. R3 can push this return traffic back into the outside service domain via this route to si-2/0/0.2. Once the return traffic is forwarded back into the service domain, the NAT can be reversed and forward the traffic back to S3.

Armed with this new information, let's verify that R3 has connectivity to S3 and S4:

```
dhanks@R3> ping 10.4.0.2 count 5 rapid routing-instance CUSTOMER_A
PING 10.4.0.2 (10.4.0.2): 56 data bytes
!!!!!
--- 10.4.0.2 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.701/1.600/3.652/1.100 ms

dhanks@R3> ping 10.5.0.2 count 5 rapid
PING 10.5.0.2 (10.5.0.2): 56 data bytes
!!!!!
--- 10.5.0.2 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.621/1.821/4.514/1.512 ms
```

Everything looks great. Now the real test is to verify that H3 can ping H4:

```
{master:0}
dhanks@H3> ping 10.5.0.10 count 5 rapid
PING 10.5.0.10 (10.5.0.10): 56 data bytes
!!!!!
--- 10.5.0.10 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max/stddev = 1.167/7.613/18.921/7.047 ms
```

The ping works and it appears that there is connectivity, but how can you be sure that the traffic was actually subject to NAT? One method is to check H4:

```
{master:0}
dhanks@H4> monitor traffic interface xe-0/1/1
verbose output suppressed, use <detail> or <extensive> for full protocol decode
Address resolution is ON. Use <no-resolve> to avoid any reverse lookup delay.
Address resolution timeout is 4s.
```

```
Listening on xe-0/1/1, capture size 96 bytes

07:29:47.274985  In IP 20.0.0.2 > 10.5.0.10: ICMP echo request
07:29:47.275024 Out IP 10.5.0.10 > 20.0.0.2: ICMP echo reply
07:29:47.276542  In IP 20.0.0.2 > 10.5.0.10: ICMP echo request
07:29:47.276568 Out IP 10.5.0.10 > 20.0.0.2: ICMP echo reply
07:29:47.295811  In IP 20.0.0.2 > 10.5.0.10: ICMP echo request
07:29:47.295853 Out IP 10.5.0.10 > 20.0.0.2: ICMP echo reply
07:29:47.308686  In IP 20.0.0.2 > 10.5.0.10: ICMP echo request
07:29:47.308723 Out IP 10.5.0.10 > 20.0.0.2: ICMP echo reply
07:29:47.311724  In IP 20.0.0.2 > 10.5.0.10: ICMP echo request
07:29:47.311758 Out IP 10.5.0.10 > 20.0.0.2: ICMP echo reply
10 packets received by filter
0 packets dropped by kernel
```

Very cool; the proof is in the pudding. Now you're able to see the translated source address of 20.0.0.2 sending a ping to 10.5.0.10. However, there is still another command that you can use on R3 to view the inline NAT pool:

```
dhanks@R3> show services inline nat pool
Interface: si-2/0/0, Service set: SS1
  NAT pool: POOL1, Translation type: BASIC NAT44
    Address range: 20.0.0.0-20.0.0.255
    NATed packets: 5, deNATed packets: 5, Errors: 0
```

The output confirms that the service set SS1 has the correct NAT pool and packets have successfully been translated. With next-hop style service sets working, let's move on to interface style and mix it up a bit.

Interface Style Service Sets

The most common form of service sets is the interface style. Just like firewall filters, the interface style service sets are applied directly to IFLs in the direction of input or output. Figure 7-10 illustrates the workflow of an interface style service set. Interface xe-2/0/1.0 has a service set applied in the input direction, whereas interface xe-7/0/0.0 has a service set applied in the output direction.

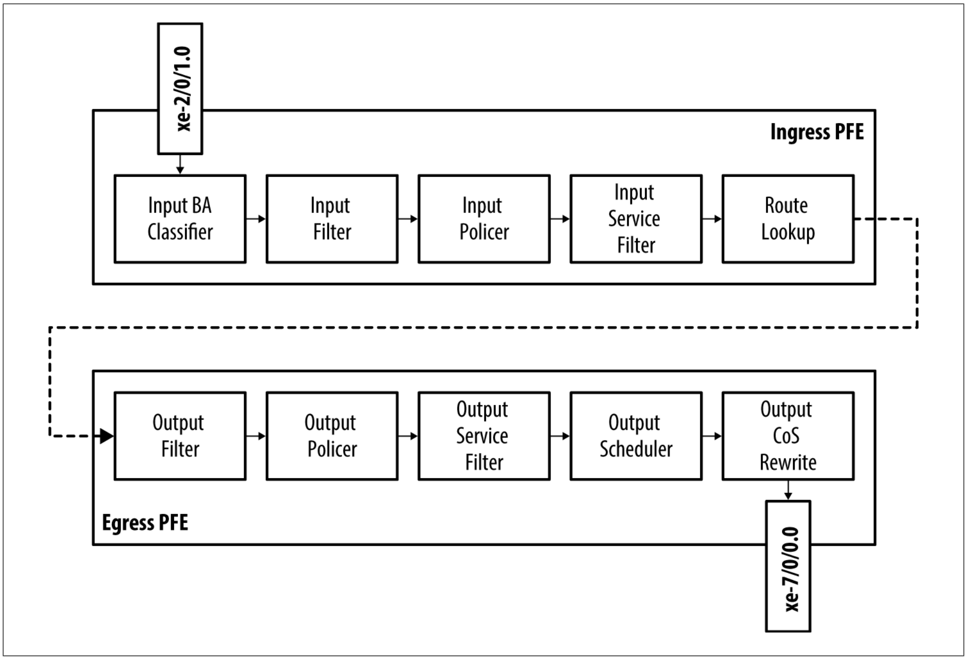


Figure 7-10. Illustration of Interface-Style Service Set Workflow.

Interface style service sets are just another “bump in the wire” from the perspective of the packet. There’s no longer the requirement to move traffic into a service interface; all that’s needed is to simply reference a service set on an IFL and specify the direction. Let’s review an example interface configuration using the interface style service sets:

```
interfaces {
    si-2/0/0 {
        unit 0 {
            family inet;
        }
    }
    xe-2/0/1 {
        unit 0 {
            family inet {
                service {
                    input {
                        service-set SS2;
                    }
                    output {
                        service-set SS2;
                    }
                }
                address 10.4.0.1/24;
            }
        }
    }
```

```
        xe-2/1/1 {
            unit 0 {
                family inet {
                    address 10.5.0.1/24;
                }
            }
        }
    }
```

There are three major points of interest. The first noticeable difference is that the services inline interface only requires the definition of unit 0 with the appropriate families it needs to process. For example, interface xe-2/0/1.0 has a family of `inet`, thus si-2/0/0.0 requires a family of `inet` as well. If a service set was applied to multiple interfaces with families of `inet` and `inet6`, then both families would need to be applied to si-2/0/0.0 as well. The second area of interest is interface xe-2/0/1. It has a service set applied to both directions. Ingress traffic will be subject to NAT on xe-2/0/1 and egress traffic will be subject to deNAT. The last interesting thing to note is the lack of a service set on xe-2/1/1. This is because the NAT and deNAT happen on a single interface. Because xe-2/0/1 is the ingress, as shown in Figure 7-11, the same interface must be used to deNAT the return traffic.

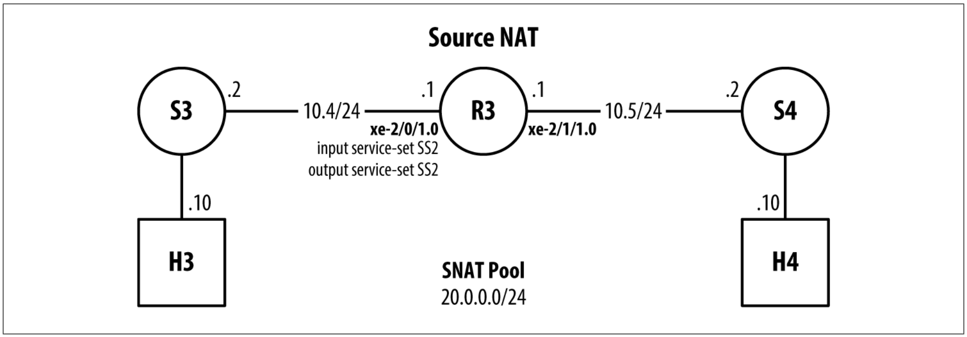


Figure 7-11. Illustration of SNAT with Interface Style Service Sets.

The traffic flow is the same as last time. Figure 7-11 illustrates that `H3` will ping 10.5.0.11 and `R3` will service the traffic via interface style service sets and translate the traffic using the 20.0.0.0/24 SNAT pool. Let's review the services configuration with interface style service sets:

```
services {
    inactive: service-set SS1 {
        nat-rules SNAT;
        next-hop-service {
            inside-service-interface si-2/0/0.1;
            outside-service-interface si-2/0/0.2;
        }
    }
    service-set SS2 {
        nat-rules SNAT;
```

```
            interface-service {
                service-interface si-2/0/0;
            }
        }
    }
```

The previous service set SS1 has been deactivated and left in the configuration for the purpose of comparison to the interface style service set SS2. Notice that the nat-rules SNAT hasn't changed, but instead using next-hop-service, you simply specify the service inline interface with the service-interface keyword. No more routing instances and input and output definitions. The NAT portion of the configuration remains the same as well. Let's review the interface style service set configuration in its entirety and test it on the topology, as shown in Figure 7-11.

```
    chassis {
        fpc 2 {
            pic 0 {
                inline-services {
                    bandwidth 1g;
                }
            }
        }
    }
    interfaces {
        si-2/0/0 {
            unit 0 {
                family inet;
            }
        }
        xe-2/0/1 {
            unit 0 {
                family inet {
                    service {
                        input {
                            service-set SS2;
                        }
                        output {
                            service-set SS2;
                        }
                    }
                    address 10.4.0.1/24;
                }
            }
        }
        xe-2/1/1 {
            unit 0 {
                family inet {
                    address 10.5.0.1/24;
                }
            }
        }
    }
    services {
        service-set SS2 {
```

```
        nat-rules SNAT;
        interface-service {
            service-interface si-2/0/0;
        }
    }
    nat {
        pool POOL1 {
            address 20.0.0.0/24;
        }
        rule SNAT {
            match-direction input;
            term 1 {
                from {
                    source-address {
                        10.4.0.0/24;
                    }
                }
                then {
                    translated {
                        source-pool POOL1;
                        translation-type {
                            basic-nat44;
                        }
                    }
                }
            }
        }
    }
}
```

With the new configuration loaded on R3, let's attempt to ping from H3 to H4 once again:

```
{master:0}
dhanks@H3> ping 10.5.0.10 rapid count 5
PING 10.5.0.10 (10.5.0.10): 56 data bytes
!!!!!
--- 10.5.0.10 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max/stddev = 1.025/3.638/8.136/3.038 ms
```

The ping was successful. Now let's verify that R3 translated the traffic by monitoring the traffic on H4:

```
{master:0}
dhanks@H4> monitor traffic interface xe-0/1/1
verbose output suppressed, use <detail> or <extensive> for full protocol decode
Address resolution is ON. Use <no-resolve> to avoid any reverse lookup delay.
Address resolution timeout is 4s.
Listening on xe-0/1/1, capture size 96 bytes

22:37:04.742892  In IP 20.0.0.2 > 10.5.0.10: ICMP echo request
22:37:04.742933 Out IP 10.5.0.10 > 20.0.0.2: ICMP echo reply
22:37:04.748307  In IP 20.0.0.2 > 10.5.0.10: ICMP echo request
22:37:04.748340 Out IP 10.5.0.10 > 20.0.0.2: ICMP echo reply
22:37:04.749876  In IP 20.0.0.2 > 10.5.0.10: ICMP echo request
22:37:04.749908 Out IP 10.5.0.10 > 20.0.0.2: ICMP echo reply
```

```
22:37:04.753714  In IP 20.0.0.2 > 10.5.0.10: ICMP echo request
22:37:04.753747 Out IP 10.5.0.10 > 20.0.0.2: ICMP echo reply
22:37:04.757974  In IP 20.0.0.2 > 10.5.0.10: ICMP echo request
22:37:04.758002 Out IP 10.5.0.10 > 20.0.0.2: ICMP echo reply
10 packets received by filter
0 packets dropped by kernel
```

Perfect. You can see that the source address from the perspective of H4 is 20.0.0.2. The last step of verification is to look at the NAT pool on R3:

```
dhanks@R3> show services inline nat pool
Interface: si-2/0/0, Service set: SS2
  NAT pool: POOL1, Translation type: BASIC NAT44
    Address range: 20.0.0.0-20.0.0.255
    NATed packets: 5, deNATed packets: 5, Errors: 0
```

Just as expected; five packets processed in each direction in the service set SS2. Everything is in working order!

Traffic Directions

With next-hop style and interface style service sets out of the way, let's circle back on traffic directions. Each style uses a different method to determine the direction of traffic. Recall that next-hop style requires that traffic be forwarded into the service interface as if it were a point-to-point tunnel; there's an inside and outside interface. The interface style works with service sets just like a firewall filter and the direction is specified directly on the IFL.

Next-Hop Style Traffic Directions. When the service interface processes a next-hop style service set, it considers traffic direction from the perspective of the service interface's inside interface. Therefore, it considers traffic received on the outside interface to be output traffic, and it considers traffic received on the inside interface to be input traffic.

Interface Style Traffic Directions. When the service interface processes an interface style service set, it considers traffic received on the interface where the service set is applied to be input traffic. Likewise, it considers traffic that is about to be transmitted on the interface to be output traffic.

One of the practical implications of this difference is that you must be careful when trying to use service rules in both interface style and next-hop style service sets. In many cases, the direction will be incorrect, and you will find that you must create different rules for use with interface style and next-hop style service sets.

Destination NAT Configuration

Destination NAT (DNAT) is similar in configuration to SNAT, but the direction of traffic is reversed and the service sets are applied on the egress interface of R3. Figure 7-12 illustrates that H3 will ping 30.0.0.10 and it will be translated to H4.

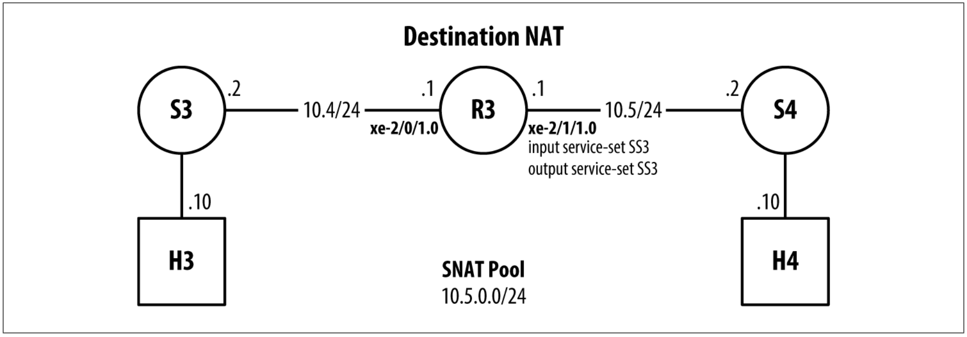


Figure 7-12. Illustration of DNAT with Interface-Style Service Sets.

The other interesting thing to note is that previously with SNAT it required a pool of source addresses to choose from as it performed the translation; with DNAT, the opposite is true. Instead of a pool of source addresses to choose from, DNAT requires a pool of destination addresses that the incoming traffic is to be translated to. For example, if `H3` pinged the address 30.0.0.10, it should be translated to 10.5.0.10. Let's check out the configuration:

```
interfaces {
    si-2/0/0 {
        unit 0 {
            family inet;
        }
    }
    xe-2/0/1 {
        unit 0 {
            family inet {
                address 10.4.0.1/24;
            }
        }
    }
    xe-2/1/1 {
        unit 0 {
            family inet {
                service {
                    input {
                        service-set SS3;
                    }
                    output {
                        service-set SS3;
                    }
                }
                address 10.5.0.1/24;
            }
        }
    }
}
services {
    service-set SS3 {
        nat-rules DNAT;
        interface-service {
            service-interface si-2/0/0;
        }
    }
    nat {
        pool POOL1 {
            address 10.5.0.0/24;
        }
        rule DNAT {
            match-direction output;
            term 1 {
                from {
                    source-address {
                        10.4.0.0/24;
                    }
                    destination-address {
                        30.0.0.0/24;
                    }
                }
                then {
                    translated {
                        destination-pool POOL1;
                        translation-type {
                            dnat-44;
                        }
                    }
                }
            }
        }
    }
}
```

The first thing to note is that the service sets have been moved to the interface on R3 that's facing the destination NAT pool of 10.5.0.0/24. Previously with the SNAT configuration, it was on the ingress interface that was facing the source NAT pool. One major difference is the **match-direction**; when using DNAT, it should be output. When H3 pings 30.0.0.10, it will ultimately egress R3 on xe-2/1/1 and be forwarded toward the destination NAT pool; thus, because the traffic is leaving the interface xe-2/1/1, the direction is **output**.

The next step is to correctly configure the match conditions for rule DNAT. The source address will be anything on the left side of 10.4.0.0/24, and the destination address will be 30.0.0.0/24. The next step is to configure the then *term* correctly and make sure the translation type is **dnat-44** and to reference the destination pool POOL1.

Let's see if H3 can ping 30.0.0.10. According to the service rules, 30.0.0.10 will be translated to the destination NAT pool of 10.5.0.0/24. Because this is a static NAT configuration, the translated address will be 10.5.0.10.

```
{master:0}
dhanks@H3> ping 30.0.0.10 count 5 rapid
PING 30.0.0.10 (30.0.0.10): 56 data bytes
```

```
!!!!!
--- 30.0.0.10 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.995/2.429/4.308/1.294 ms
```

Perfect. H3 has connectivity via DNAT to H4.

Network Address Translation Summary

Trio inline NAT is performed directly within the Lookup Block and offers near line-rate performance, but at the expense of limited functionality when compared to the MS-DPC. However, static NAT can have three variations: source NAT, destination NAT, and twice NAT. Being able to provide inline NAT services without the MS-DPC provides distinct performance and cost advantages; the icing on the cake is that the configuration style between Trio inline and MS-DPC NAT is the same.

Tunnel Services

Junos makes working with encapsulation very easy and straightforward. Tunnel services are a collection of encapsulation and decapsulation logical interfaces that are used to help forward traffic. The types of tunnel services supported are as follows:

IP Tunnel (IPIP)
: IPIP is a very basic IP tunneling protocol that simply encapsulates the original packet in a new IP header. The interface name for IPIP in Junos is "ip-."

Generic Routing Encapsulation (GRE)
: GRE improves upon IPIP and adds the ability to enforce packet sequencing, specify tunnel keys, and encapsulation any Layer 3 protocol without being limited to only IP. The interface name for GRE in Junos is "gr-."

Logical Tunnels
: Logical tunnels are pseudointerfaces in Junos; they look and feel like regular interfaces, but don't consume any physical ports. A common use case is to use a logical tunnel to interconnect two VRFs. The interface name for logical tunnels in Junos is "lt-."

Protocol Independent Multicast (PIM) Encapsulation and Decapsulation
: These interfaces are used by PIM designated routers (DR) or rendezvous points (RP) to encapsulate and decapsulate packets during the PIM JOIN and REGISTER processes. The interface names for PIM encapsulation and decapsulation in Junos are "pe-" and "pd-."

In previous platforms, a special Tunnel PIC was required to enable these interfaces. The DPC line cards also required that a single 10 G port be disabled when using tunnel services; however, with the Trio-based line cards, the tunnel services are built into each PFE and there's no loss of revenue ports. Because the tunnel service processing happens directly on the line card, the performance is near line-rate and keeps the latency to a minimum.

Enabling Tunnel Services

Because tunnel services are enabled through the line card, the scale is directly proportional to the number of PICs in the chassis. For example, the MPC1 supports one set of tunnel services while the MPC2 supports two sets of tunnel services. To create the tunnel services, you must select a FPC and PIC to bind it to:

```
chassis {
    fpc 2 {
        pic 0 {
            tunnel-services {
                bandwidth 1g;
            }
        }
    }
}
```

In this example, FPC 2 and PIC 0 are associated with an instance of tunnel services. The last option when creating tunnel services is the amount of bandwidth required; the values vary by line card, but the most common options are 1 g or 10 g for MPC1 and MPC2 line cards. The bandwidth option specifies the amount of bandwidth that will be available to the encapsulation and decapsulation logical interfaces.

A good method to determine what type of interfaces are created by tunnel services is to take a "snapshot" before and after the configuration change. Let's start by taking a snapshot of the current state:

```
[edit]
dhanks@R4# run show interfaces terse | save before-ts
Wrote 76 lines of output to 'before-ts'
```

This saves the output of `show interfaces terse` to a file called `before-ts`. This file will be used as a reference to the number of interfaces before the change. Now let's enable tunnel services and commit:

```
[edit]
dhanks@R4# set chassis fpc 2 pic 0 tunnel-services bandwidth 10g

[edit]
dhanks@R4# commit
commit complete
```

Now let's use the same method as before to capture the interface list and save it to a file called `after-ts`:

```
[edit]
dhanks@R4# run show interfaces terse | save after-ts
Wrote 84 lines of output to 'after-ts'
```

Perfect. Now there is a snapshot before and after enabling tunnel services. Let's use the `file compare` command to view the differences:

```
[edit]
dhanks@R4# run file compare files before-ts after-ts
> gr-2/0/0                up    up
> ip-2/0/0                up    up
> lt-2/0/0                up    up
> mt-2/0/0                up    up
> pd-2/0/0                up    up
> pe-2/0/0                up    up
> ut-2/0/0                up    up
> vt-2/0/0                up    up
```

Eight new interfaces have been created after enabling tunneling services on FPC 2 and PIC 0. The interface naming convention is the typical `name-FPC/PIC/PORT`. For example, the GRE interface is `gr-2/0/0`. If FPC2 and PIC 1 were to be used instead, the name would have been `gr-2/1/0`. The port number is tied to the amount of bandwidth associated with the tunnel services. The general rule of thumb is that `bandwidth` of 1g has a port number of 10 while all other values higher than 1g have a port number of 0.

Table 7-2. Tunnel Services Bandwidth Port Assignment.

Bandwidth	Port Number
1g	10
10g	0
20g	0
30g	0
40g	0
65g	0

The `bandwidth` knob is optional. If a bandwidth isn't specified, it will default to the highest possible setting. For MPC1 and MPC2, this will be 10g, and MPC3E will be 65g.

Given that the scale of tunnel services is directly proportional to the number of PICs, it's possible to have 48 instances of tunnel services using 480 Gbps of bandwidth, assuming a fully loaded Juniper MX960 with 12 MPC-3D-16x10GE line cards. Each MPC-3D-16x10GE line card has four PICs, thus 48 instances of tunnel services.

Tunnel Services Case Study

Having the ability to have tunnel services at your fingertips without having to purchase additional hardware gives you the instant flexibility to solve interesting problems and create interesting topologies. Let's create a case study that uses a couple of different tunnel services interfaces to forward traffic between routers.

This case study will use a mixture of logical systems, logical tunnels, and GRE tunnels in a multiarea OSPF topology. Logical systems are simply a router within a router. A logical system has its own interfaces, configuration, and routing protocols. It's an easy way to simulate a completely separate router without the additional hardware. The reason this case study uses logical systems is that the logical router needs a method to communicate with the physical router; this is where logical tunnels come in. Logical tunnels can be paired together and create a virtual Ethernet cable between the two routers.

The only physical routers in this case study are R1 and R3. Each router will have its own logical system defined for a total of four routers: R1, R3, LS1, and LS3. Physical interfaces such as xe-2/0/1 will connect R1 and R4, whereas logical tunnels will connect R1 and LS1 and R3 to LS3, as shown in Figure 7-13.

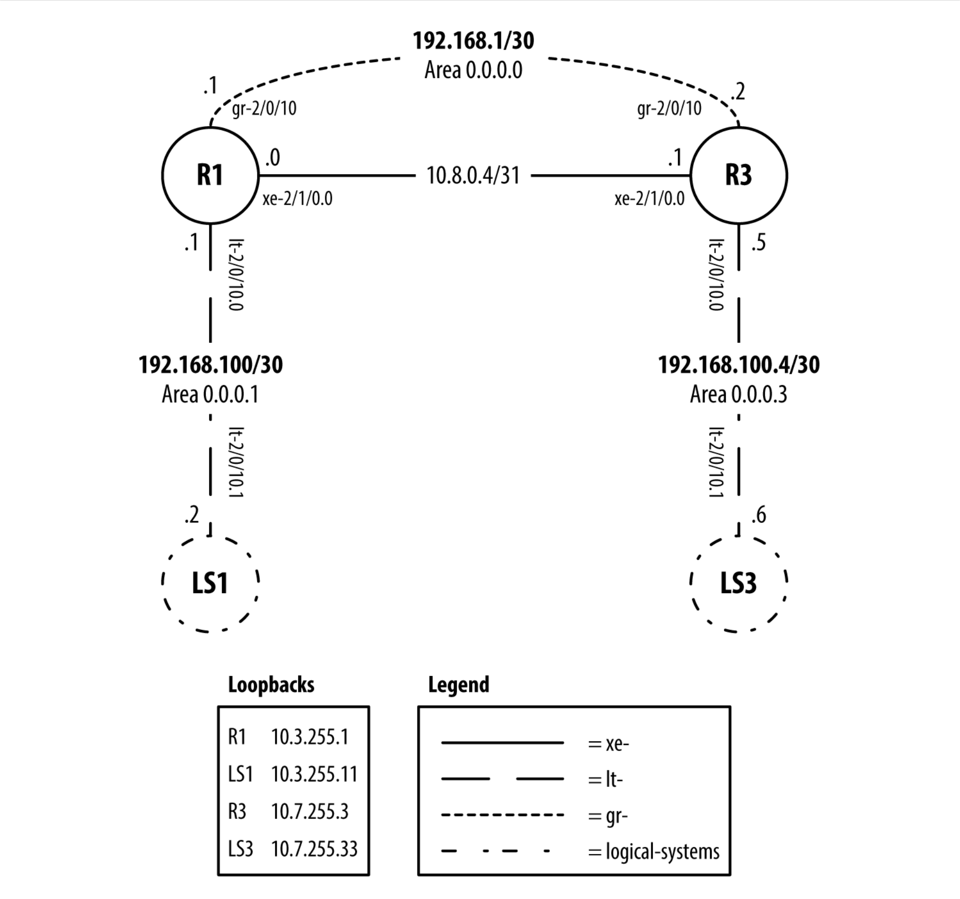


Figure 7-13. Tunnel Services Case Study.

Let's start from the top. R1 and R3 are directly connected via xe-2/0/1 on the 10.8.0.4/31 network. These physical interfaces do not participate in any sort of routing protocol:

```
interfaces {
    xe-2/1/0 {
        unit 0 {
            family inet {
                address 10.8.0.4/31;
            }
        }
    }
    lo0 {
        unit 0 {
            family inet {
                address 10.3.255.1/32;
            }
        }
}
```

This use case assumes there is some type of "network" between R1 and R3 that is out of your control, and the only method to create a direct link between R1 and R3 is via a GRE tunnel. Let's review the GRE interface configuration on R1:

```
interfaces {
    gr-2/0/10 {
        unit 0 {
            tunnel {
                source 10.8.0.4;
                destination 10.8.0.5;
            }
            family inet {
                address 192.168.1.1/30;
            }
        }
    }
}
```

There are two things required for the creation of a GRE tunnel: the source and destination addresses of the two ends points creating the tunnel, and the actual addressing information the tunnel will carry. As shown in Figure 7-13, the GRE tunnel between R1 and R3 has the network 192.168.1.0/30.

Let's verify that the GRE tunnel is up and has connectivity between R1 and R3:

```
dhanks@R1-RE0> ping count 5 rapid 192.168.100.2
PING 192.168.100.2 (192.168.100.2): 56 data bytes
!!!!!
--- 192.168.100.2 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.554/0.580/0.673/0.046 ms
```

Perfect; the gr-2/0/10.0 interface is up and forwarding traffic. Another great feature of the Trio chipset is that it allows for inline operations, administration, and maintenance (OAM) keepalive messages across the GRE tunnel:

```
protocols {
    oam {
        gre-tunnel {
            interface gr-2/0/10 {
                keepalive-time 1;
                hold-time 5;
            }
        }
    }
}
```

This configuration is required on each end of the GRE tunnel. The only options are the keepalive time and hold timer. The **keepalive-time** is the time in seconds between keepalive messages, and the **hold-time** determines how many seconds have to pass without receiving a keepalive to render the neighbor dead. In this case, five keepalive messages have to be missed before the tunnel is considered down. Let's use the **show oam** command to verify that the GRE keepalive messages are being sent and received:

```
dhanks@R1-RE0> show oam gre-keepalive interface-name gr-2/0/10.0
Interface name       Sent       Received       Status
  gr-2/0/10.0       953         953         tunnel-oam-up
```

Now that the GRE tunnel is up on 192.168.1.100/30 and has been verified with OAM, the next step to building the OSPF network is to define the backbone area between R1 and R3. R1 and R3 will peer via OSPF in area 0.0.0.0 across the GRE tunnel:

```
protocols {
    ospf {
        reference-bandwidth 100g;
        area 0.0.0.0 {
            interface gr-2/0/10.0 {
                interface-type p2p;
                bfd-liveness-detection {
                    minimum-interval 150;
                    multiplier 3;
                }
            }
            interface lo0.0 {
                passive;
            }
        }
    }
}
```

The GRE interface gr-2/0/10.0 has been placed into OSPF area 0.0.0.0 on R1 and R3. One trick to speed up the convergence when using point-to-point links is to determine the **interface-type** as **p2p**. This will simply bypass the DR and BDR election process. BFD will also be used across the GRE tunnel to quickly bring down the OSPF neighbor upon the detection of a forwarding failure.

At this point R1 and R3 should have OSPF and BFD up and operational. Let's verify from the perspective of R1:

```
dhanks@R1-RE0> show ospf neighbor
Address          Interface              State     ID               Pri  Dead
192.168.1.2      gr-2/0/10.0            Full      10.7.255.3       128    37

dhanks@R1-RE0> show bfd session
                                                  Detect   Transmit
Address                  State     Interface      Time     Interval  Multiplier
192.168.1.2              Up        gr-2/0/10.0    0.450     0.150        3

1 sessions, 1 clients
Cumulative transmit rate 6.7 pps, cumulative receive rate 6.7 pps
```

OSPF and BFD are up. Let's check connectivity between R1 and R3 by sourcing a ping from R1's loopback and using R3's loopback as the destination:

```
dhanks@R1-RE0> ping count 5 rapid source 10.3.255.1 10.7.255.3
PING 10.7.255.3 (10.7.255.3): 56 data bytes
!!!!!
--- 10.7.255.3 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.575/0.644/0.789/0.088 ms
```

Perfect. At this point GRE, OSPF, and BFD are up and able to forward traffic. The next step is to build out the logical systems on R1 and R3. The configuration is very easy and only requires the definition of interfaces and logical-systems:

```
interfaces {
    lt-2/0/10 {
        unit 0 {
            encapsulation ethernet;
            peer-unit 1;
            family inet {
                address 192.168.100.1/30;
            }
        }
        unit 1 {
            encapsulation ethernet;
            peer-unit 0;
            family inet {
                address 192.168.100.2/30;
            }
        }
    }
    lo0 {
        unit 0 {
            family inet {
                address 10.3.255.1/32;
            }
        }
        unit 1 {
            family inet {
                address 10.3.255.11/32;
            }
        }
    }
}
```

There are two new interfaces being added: logical tunnels and an additional loopback address to be assigned to the new logical system. Logical tunnels are defined in pairs, as they act as a virtual Ethernet wire inside of the router. The logical tunnel lt-2/0/10.0 will be assigned to R1, whereas lt-2/0/10.1 will be assigned to the logical system LS1. The glue that ties these two IFLs together is the knob peer-unit. Each IFL needs to point to the other IFL it wants to pair with. For example, lt-2/0/10.0 peer-unit points to unit 1 and lt-2/0/10.1 peer-unit points to unit 0. Now the logical tunnel is built and the two units are able to directly communicate via this virtual wire.

Because LS1 will be a new virtual router that will participate in OSPF, it will require its own dedicated loopback to use as the OSPF router ID. The interface lo0.1 was created and assigned the address 10.3.255.11/32. The next step is to create the logical system LS1 and associate the interfaces with it:

```
logical-systems {
    LS1 {
        interfaces {
            lt-2/0/10 {
                unit 1;
            }
            lo0 {
                unit 1;
            }
        }
    }
}
```

In Junos, only a single loopback IFL can exist in a routing instance or logical system. When creating the new LS1 logical system, a new IFL is required, thus lo0.1. In situations where there need to be multiple loopbacks in the same routing instance or logical system, simply add another IFA to the loopback IFL. For example, lo0.0 could have 100 IPv4 addresses in the master routing instance.

Now the logical system LS1 has been created and has been assigned two interfaces: lo0.1 and lt-2/0/10.1. Let's check the connectivity across the logical tunnel by sourcing a ping from R1 and using LS1 as the destination via the 192.168.100/30 network:

```
dhanks@R1-RE0> ping count 5 rapid 192.168.100.2
PING 192.168.100.2 (192.168.100.2): 56 data bytes
!!!!!
--- 192.168.100.2 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.554/0.580/0.673/0.046 ms
```

Excellent. Now that R1 and LS1 have connectivity, the next step is to configure OSPF on R1 and LS1 across the logical tunnel and place it into OSPF area 0.0.0.1. Let's start with R1:

```
1    protocols {
2        ospf {
3            reference-bandwidth 100g;
4            area 0.0.0.0 {
5                interface gr-2/0/10.0 {
6                    interface-type p2p;
7                    bfd-liveness-detection {
8                        minimum-interval 150;
9                        multiplier 3;
10                    }
11                }
12                interface lo0.0 {
13                    passive;
14                }
15            }
16            area 0.0.0.1 {
17                interface lt-2/0/10.0 {
18                    interface-type p2p;
19                }
20            }
21        }
22    }
```

Lines 16 through 20 show the addition of OSPF area 0.0.0.1 and the logical tunnel that connect `R1` to `LS1`. Now let's configure OSPF on `LS1`:

```
logical-systems {
    LS1 {
        protocols {
            ospf {
                reference-bandwidth 100g;
                area 0.0.0.1 {
                    interface lt-2/0/10.1 {
                        interface-type p2p;
                    }
                    interface lo0.1 {
                        passive;
                    }
                }
            }
        }
    }
}
```

The logical system `LS1` has now been configured for OSPF in area 0.0.0.1 on the logical tunnel lt-2/0/10.1. Let's verify that OSPF has discovered its new neighbor:

```
dhanks@R1-RE0> show ospf neighbor
Address          Interface              State     ID               Pri  Dead
192.168.1.2      gr-2/0/10.0            Full      10.7.255.3       128    37
192.168.100.2    lt-2/0/10.0            Full      10.3.255.0       128    35
```

Very cool. `R1` is now showing both the GRE tunnel and logical tunnel in an OSPF state of Full. However, let's take a closer look at the logical systems before moving on. It's true that logical systems act as a virtual router, but Junos has a few tricks up its sleeve.

The `set cli logical-system` command will modify the CLI to operate from the perspective of the referenced logical system. Let's try changing the CLI and login to `LS1`:

```
dhanks@R1-RE0> set cli logical-system LS1
Logical system: LS1

dhanks@R1-RE0:LS1> show ospf neighbor
Address          Interface              State     ID               Pri  Dead
192.168.100.1    lt-2/0/10.1            Full      10.3.255.1       128    32
```

Interesting! Now every command executed will operate and respond as if you're logged into the logical system. Let's test out this theory some more. What about `show route`?

```
dhanks@R1-RE0:LS1> show route

inet.0: 7 destinations, 7 routes (7 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

10.3.255.0/32      *[Direct/0] 00:11:12
                    > via lo0.1
10.3.255.1/32      *[OSPF/10] 00:05:55, metric 100
                    > to 192.168.100.1 via lt-2/0/10.1
10.3.255.11/32     *[Direct/0] 00:11:12
                    > via lo0.1
192.168.1.0/30     *[OSPF/10] 00:05:55, metric 200
                    > to 192.168.100.1 via lt-2/0/10.1
192.168.100.0/30   *[Direct/0] 00:11:11
                    > via lt-2/0/10.1
192.168.100.2/32   *[Local/0] 00:11:11
                      Local via lt-2/0/10.1
224.0.0.5/32       *[OSPF/10] 00:11:12, metric 1
                      MultiRecv
```

I'm sorry, but that's way too cool! Even the route table looks and feels as if you're logged into another router. Given this new CLI tool, it makes using logical systems a breeze. To logout of the router and go back to the physical router type:

```
dhanks@R1-RE0:LS1> clear cli logical-system
Cleared default logical system

dhanks@R1-RE0>
```

At this point, the left side of the topology as illustrated in Figure 7-13 has been configured and tested. The only remaining tasks are to replicate the logical tunnels and logical systems on `R3` and `LS3`. The only difference is that the right side of the topology will use OSPF area 0.0.0.3 and the logical tunnel will use the 192.168.1.4/30. This case study will skip the configuration and verification of the connectivity between `R3` and `LS3` because it's nearly identical to `R1` and `LS1`.

Tunnel Services Case Study Final Verification

Once R3 and LS3 have been configured and verified using the same process used with R1 and LS1, there should be complete end-to-end connectivity, as illustrated in Figure 7-14.

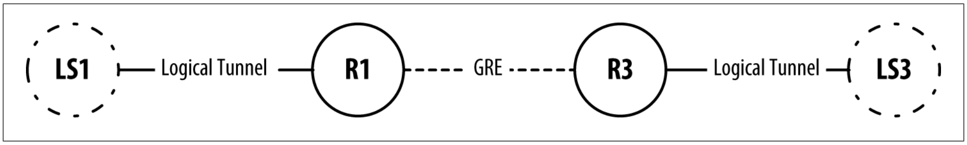


Figure 7-14. Tunnel Services Case Study: Logical End-to-End Connectivity.

Each router is connected by a logical interface that is part of the tunnel services offered by the Juniper MX. Let's login to LS1 and verify the route table and see if the R3 and LS3 routes show up:

```
dhanks@R1-RE0> set cli logical-system LS1
Logical system: LS1

dhanks@R1-RE0:LS1> show route

inet.0: 10 destinations, 10 routes (10 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

10.3.255.0/32      *[Direct/0] 00:11:12
                    > via lo0.1
10.3.255.1/32      *[OSPF/10] 00:05:55, metric 100
                    > to 192.168.100.1 via lt-2/0/10.1
10.3.255.11/32     *[Direct/0] 00:11:12
                    > via lo0.1
10.7.255.3/32      *[OSPF/10] 00:05:55, metric 200
                    > to 192.168.100.1 via lt-2/0/10.1
10.7.255.33/32     *[OSPF/10] 00:04:00, metric 300
                    > to 192.168.100.1 via lt-2/0/10.1
192.168.1.0/30     *[OSPF/10] 00:05:55, metric 200
                    > to 192.168.100.1 via lt-2/0/10.1
192.168.100.0/30   *[Direct/0] 00:11:11
                    > via lt-2/0/10.1
192.168.100.2/32   *[Local/0] 00:11:11
                      Local via lt-2/0/10.1
192.168.100.4/30   *[OSPF/10] 00:05:55, metric 300
                    > to 192.168.100.1 via lt-2/0/10.1
224.0.0.5/32       *[OSPF/10] 00:11:12, metric 1
                      MultiRecv
```

Very cool. Both R3 (10.7.255.3) and LS3 (10.7.255.33) are in the route table. For the final step, let's verify that LS1 (10.3.255.11) has connectivity to LS3 (10.7.255.33):

```
dhanks@R1-RE0:LS1> ping rapid count 5 source 10.3.255.11 10.7.255.33
PING 10.7.255.33 (10.7.255.33): 56 data bytes
!!!!!
--- 10.7.255.33 ping statistics ---
```

```
    5 packets transmitted, 5 packets received, 0% packet loss
    round-trip min/avg/max/stddev = 0.591/0.683/1.037/0.177 ms
```

Everything works as expected and LS1 has full connectivity to LS2. It's amazing to consider that such a topology is possible and operate at near line rate via tunnel services that originate from the line cards themselves. Previously, such features required a dedicated Tunnel PIC and couldn't offer the same performance.

Tunnel Services Summary

The Juniper MX packs a mean punch. Being able to offer GRE, IPIP, PIM, and logical tunnels without a separate piece of hardware offers a significant advantage. Because the Trio chipset processes the tunnel services and encapsulation/decapsulation in the Lookup Block, the performance is near line rate and reduces the latency when compared with having to send the packet to a services card and back.

The scale of tunnel services is directly proportional to the number of Trio Lookup Blocks in a chassis. The amount of bandwidth used by tunnel services can be reserved from 1g to 65g depending on the line card. The beauty of the Trio chipset is that enabling tunnel services doesn't waste a WAN port, because all of the processing is in the Trio Lookup Block.

Port Mirroring

One of the most useful tools in the troubleshooting bag is port mirroring. It allows you to specify traffic with a firewall filter and copy it to another interface. The copied traffic can then be used for analysis or testing. One of the most interesting use cases I recall is when a customer wanted to test a firewall's throughput on production data, but obviously not impact production traffic. Port mirroring was the perfect tool to match the production data and send a copy of the traffic to the firewall under test, while the original traffic was forwarded to its final destination on the production network.

Junos has supported port mirroring for a very long time, and the architecture is very simple and flexible. There are four major components that make up port mirroring, as shown in Figure 7-15: FPC, port mirroring instances, next-hop groups, and next-hops.

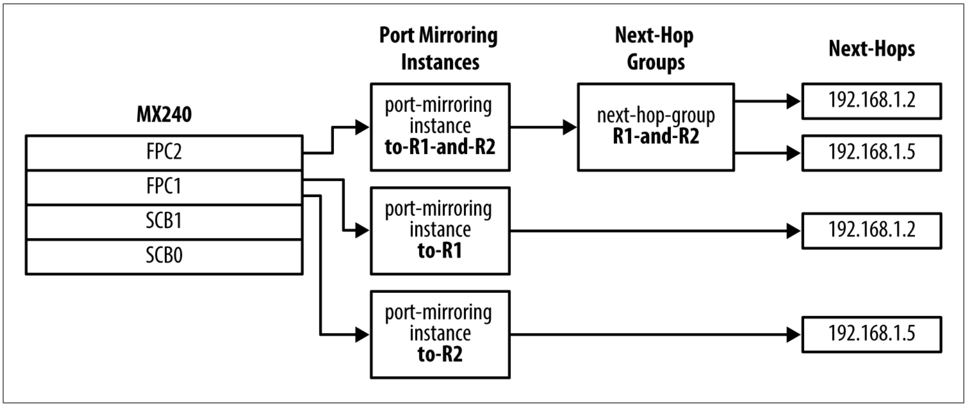


Figure 7-15. Port Mirroring Workflow.

Port mirroring instances are associated with FPCs, and up to two port mirroring instances can be associated with a single FPC. This concept is similar to other Trio inline functions where the FPC is associated to an instance or inline service. The next component is a next-hop group; this is simply a collection of interfaces and associated next-hops. The use of a next-hop group is optional. The last components are the next-hops that reference a specific interface and next-hop.

However, no traffic will be subject to port mirroring until there's a firewall filter to match traffic and send a copy of it to the port mirroring instance, as illustrated in Figure 7-16.

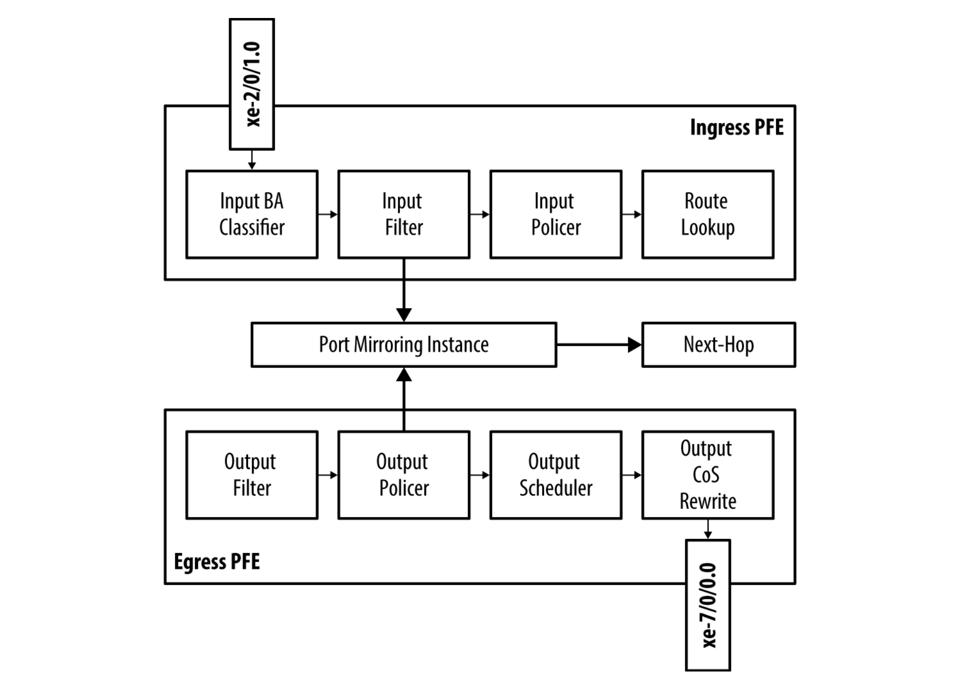


Figure 7-16. Firewall Filter Matching Traffic for Port Mirroring.

Because regular firewall filters are used, port mirroring works either as an input or output filter. This creates the possibility of copying the same traffic to the port mirroring instance twice. For example, if the same firewall filter was applied to interface xe-2/0/1.0 as an input filter and to the interface xe-7/0/0.0 as an output filter, the same traffic will be matched and sent to the port mirroring instance. In such scenarios, Junos offers an option to "mirror once" and ignore the duplicate match and only send a single copy into the port mirroring instance.

Port Mirror Case Study

With the basics out of the way, let's get down to business and learn how to configure the different components of port mirroring and create an interesting case study, as illustrated in Figure 7-17. The first step is to generate traffic that can be used for port mirroring. The traffic flow will be basic IPv4 from S4 to S3. IS-IS is configured on all routers for reachability. Traffic sourced from S4 and destined to S3 would put R3 in the middle of the path, which is a perfect spot to set up port mirroring. Using the power of a traffic generator, the author was able to generate 10,000 PPS from sourced from S4 and destined to S3.

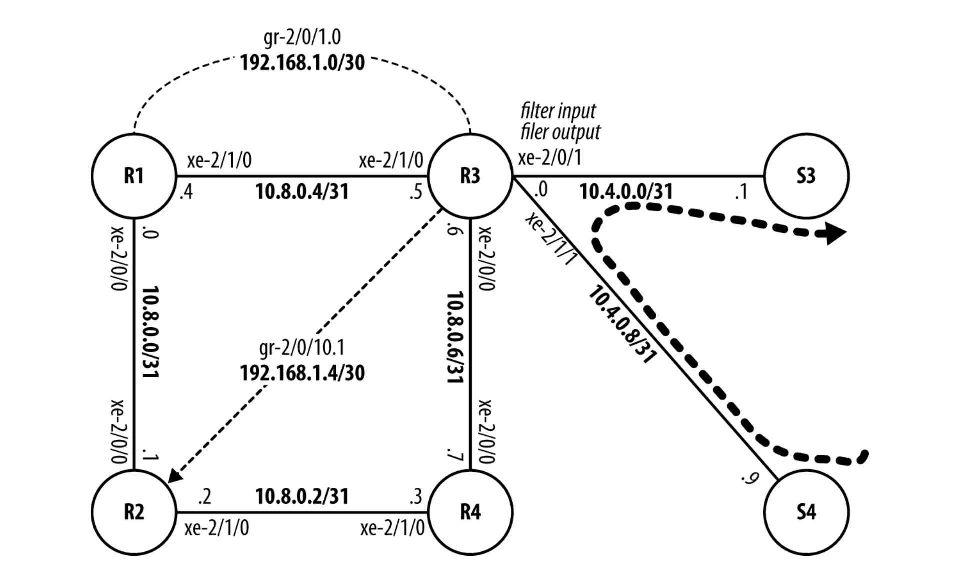


Figure 7-17. Case Study: Port Mirroring with Next-Hop Groups and GRE.

There are two GRE tunnels defined between R3 and R1 and between R3 and R2. These GRE interfaces and next-hops will be used as part of the port mirroring instances. The flexibility of Junos really shines when it comes to next-hops and port mirroring. It's a common misconception that a port mirror can only copy traffic to a local interface, but this isn't true. Using GRE tunnels, it's possible to port mirror traffic across the data center or the Internet to a remote Linux server.

Configuration

The first step is to configure the port mirroring instance, which is located in the [forwarding-options] hierarchy of the configuration. The port mirroring instance also needs to be associated with an FPC. In this example, the port mirroring instance will be associated with FPC 2:

```
chassis {
    fpc 2 {
        port-mirror-instance to-R1-and-R2;
    }
}
forwarding-options {
    port-mirroring {
        instance {
            to-R1-and-R2 {
                input {
                    rate 1;
                }
                family inet {
                    output {
                        next-hop-group R1-and-R2;
                    }
                }
            }
        }
    }
}
```

There are only two components required when defining a port mirroring instance: input and output. The input specifies the sampling rate, using the same formula as J-Flow sampling, as illustrated in Figure 7-18.

$$\boxed{\text{packets\_sampled} = \frac{\text{run-length} + 1}{\text{rate}}}$$

Figure 7-18. Port Mirroring Input Rate Formula.

In the case study configuration, the `run-length` is omitted and the `rate` is set to 1; this will mirror every single packet that's copied into the port mirroring instance.

The next component is the output; this specifies the interface and next-hop to be used to send the port mirror traffic to. The case study uses a next-hop group called `R1-and-R2`:

```
forwarding-options {
    next-hop-group R1-and-R2 {
        group-type inet;
        interface gr-2/0/10.0 {
            next-hop 192.168.1.1;
        }
        interface gr-2/0/10.1 {
            next-hop 192.168.1.5;
        }
    }
}
```

Here, the individual interfaces and next-hops are defined. The interface gr-2/0/10.0 and next-hop of 192.168.1.1 will send mirrored traffic to `R1`, and the interface gr-2/0/10.1 and next-hop of 192.168.1.5 will send the mirrored traffic to `R2`. Let's verify the creation of the next-hop group:

```
dhanks@R3> show forwarding-options next-hop-group detail
Next-hop-group: R1-and-R2
  Type: inet
  State: up
  Number of members configured    : 2
  Number of members that are up   : 2
  Number of subgroups configured  : 0
  Number of subgroups that are up : 0
  Members Interfaces:                                State
    gr-2/0/10.0            next-hop  192.168.1.1     up
    gr-2/0/10.1            next-hop  192.168.1.5     up
```

The next-hop group `R1-and-R2` is showing two members configured with the correct interfaces and next-hops. Let's check to see if the port mirroring instance is available:

```
dhanks@R3> show forwarding-options port-mirroring

Instance Name: to-R1-and-R2
  Instance Id: 5
  Input parameters:
    Rate                  : 1
    Run-length            : 0
    Maximum-packet-length : 0
  Output parameters:
    Family      State     Destination          Next-hop
    inet        up        R1-and-R2
```

The port mirroring instance `to-R1-and-R2` is showing the correct input and output vales as well. Everything is looking good so far. If you are feeling pedantic, there is a shell command to verify that the port mirroring instance has been installed into the PFE on FPC2:

```
dhanks@R3>
request pfe execute target fpc2 command "show sample instance association"
SENT: Ukern command: show sample instance association
GOT:
GOT: Sampler Parameters
GOT: Global Sampler Association: "&global_instance"
GOT: FPC Bindings              :
GOT: sampling    :
GOT: port-mirroring 0 : to-R1-and-R2
GOT: port-mirroring 1 :
GOT: PIC[0]Sampler Association:
GOT: sampling    :
GOT: port-mirroring 0 :
GOT: port-mirroring 1 :
GOT: PIC[1]Sampler Association:
GOT: sampling    :
GOT: port-mirroring 0 :
GOT: port-mirroring 1 :
GOT: Sampler Association
GOT: PFE[0]-[0]Sampler Association: "to-R1-and-R2":class 2 proto 0 instance id 5
GOT: PFE[1]-[0]Sampler Association: "to-R1-and-R2":class 2 proto 0 instance id 5
LOCAL: End of file
```

Using shell commands comes with the usual disclaimer of "do not use this in production."

The port mirror instance to-R1-and-R2 is now associated to FPC2 as shown by lines 18 through 20. The final piece of the puzzle is to create a firewall filter to match traffic and place a copy into the port mirroring instance:

```
firewall {
    family inet {
        filter mirror-next-hop-group-R1-and-R2 {
            term 1 {
                then port-mirror-instance to-R1-and-R2;
            }
        }
    }
}
```

The firewall filter mirror-next-hop-group-R1-and-R2 simply matches all traffic and sends a copy to the port mirroring instance to-R1-and-R2. As shown in Figure 7-17, this filter will be applied to R3's interface xe-2/0/1 in both input and output directions. As traffic is flowing between S3 and S4, R3 will be able to match the transit traffic with the firewall filter then send a copy of the matched traffic to both GRE tunnels which are destined to R1 and R2.

Let's take a peek at the packets per second on R3's interface xe-2/0/1 to see how much traffic is flowing through:

```
dhanks@R3> show interfaces xe-2/0/1 | match pps
  Input rate     : 6659744 bps (9910 pps)
  Output rate    : 6659744 bps (9910 pps)
```

Not bad; there's about 10,000 PPS running through R3 on the interface facing S3. One good method to check and see if port mirroring is working is to check the packets per second on the next-hop group or output interfaces:

```
dhanks@R3> show interfaces gr-2/0/10 | match pps
  Input rate     : 0 bps (0 pps)
  Output rate    : 19546016 bps (29086 pps)
```

Wait a second, nearly 30,000 PPS on the output GRE tunnel is way too much traffic. Something is wrong. Let's run a simple ping command on S4 to double check the connectivity to S3 (10.4.0.1):

```
dhanks@S4> ping 10.4.0.1
PING 10.4.0.1 (10.4.0.1): 56 data bytes
64 bytes from 10.4.0.1: TTL expired in transit.
64 bytes from 10.4.0.1: icmp_seq=0 ttl=64 time=0.737 ms (DUP!)
64 bytes from 10.4.0.1: icmp_seq=0 ttl=64 time=0.752 ms (DUP!)
64 bytes from 10.4.0.1: icmp_seq=0 ttl=64 time=0.764 ms (DUP!)
^C
```

Feeling a bit loopy there. What happened? Recall that the entire topology is running IS-IS for reachability and that port mirroring is sending a copy of every packet down to GRE tunnels destined for R1 and R2. Both R1 and R2 receive a copy of the packet and forward it like a regular packet. In this case, the packet is destined to S4, so both R1 and R2 will forward back to S4.

This obviously creates a routing loop. It's important to remember that the device on the other end of the port mirror must not have the ability to forward the mirrored traffic, otherwise it will quickly cause problems on your network. The easiest solution to this problem is to create an empty routing instance on R1 and R2 to house the GRE tunnel. When R1 and R2 receive the mirrored packets, the new routing instance will have an empty route table and just drop the packets.

With this routing instance solution in place on R1 and R2, let's double check the connectivity on S4 again:

```
dhanks@S4> ping count 5 rapid 10.4.0.1
PING 10.4.0.1 (10.4.0.1): 56 data bytes
!!!!!
--- 10.4.0.1 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.575/0.644/0.789/0.088 ms
```

Much better. No packet loss, no duplicates, and no loops! Let's get back to the original task and check the PPS on the port mirror instance interfaces:

```
{master}
dhanks@R3-RE0> show interfaces gr-2/0/10 | match pps
  Input rate     : 13319488 bps (19820 pps)
  Output rate    : 0 bps (0 pps)
```

The two GRE tunnels on R3 are definitely sending traffic at the same speed as being mirrored on interface xe-2/0/1 on. Recall that interface xe-2/0/1 was measured at 9910 PPS. Because a copy of each packet received on interface xe-2/0/1 will be sent to both GRE tunnels gr-2/0/10.0 (to R1) and gr-2/0/10.1 (to R2), the aggregate PPS on gr-2/0/10 makes sense.

Port Mirror Summary

Port mirroring is a very powerful and flexible tool that grants you the ability to perform either local or remote traffic analysis. Always ensure that the device receiving the mirrored traffic doesn't have a forwarding path to the destinations in the mirrored traffic, otherwise a routing loop will occur and cause problems in the network.

Coupled with firewall filters, it's possible to only mirror a certain subset of traffic such as:

- Only HTTP traffic
- Only traffic from a specific customer
- Only traffic from a specific malicious user or botnet

This gives you surgical control over what traffic to port mirror; when performing traffic analysis, the selective port mirroring will allow for faster data processing because uninteresting traffic has already been filtered out.

Summary

The Trio chipset offers line rate performance with the ability to offer inline services. As new versions of Junos are released, the Trio chipset has the ability to be upgraded and offer additional services as well. This new architecture of offering inline services in the Trio chipset frees you from having to invest in additional hardware for service processing. However, the Trio inline services are basic in nature as of Junos 11.4, and any advanced features such as carrier-grade NAT (CGN) requires the MS-DPC.

It's amazing to consider that inline services such as J-Flow, NAT, GRE, and port mirroring are available from the line card itself. One of the big benefits of keeping the services within the Trio chipset is low-latency and near line rate performance. This is because the packet doesn't have to travel to a services card, be processed, and sent back to the line card; everything is done locally within the Trio chipset.

Chapter Review Questions

1. Which versions of J-Flow are supported in Trio inline services as of Junos 11.4?
   a. J-Flow v5
   b. J-Flow v9
   c. J-Flow v10
   d. IPFIX
2. What types of NAT are not supported in Trio inline services as of Junos 11.4?
   a. SNAT
   b. SNAPT
   c. Twice NAT
   d. DNAPT
3. Which service set type allows you to bridge two routing instances?
   a. Next-hop style
   b. Interface style
4. Which interfaces are not part of Trio tunnel services?
   a. gre-0/0/0
   b. lt-2/0/10
   c. vt-3/0/0
   d. ge-0/0/0
5. What binds together logical tunnels?
   a. Network address
   b. Circuit ID

    c. Peer unit
    d. Encapsulation type
6. How many port mirroring instances can be associated with a FPC?
    a. 1
    b. 2
    c. 3
    d. 4
7. What percentage of packets would be sampled with a `rate` of 10 and `run-length` of 3?
    a. 10%
    b. 30%
    c. 40%
    d. 100%

Chapter Review Answers

1. **Answer: C,D**. J-Flow v10 and IPFIX are synonymous. Trio supports both.
2. **Answer: B,D**. As of Junos 11.4, Trio only supports 1:1 SNAT, DNAT, and Twice NAT.
3. **Answer: A**. The next-hop style service set creates an inside and outside service interface that can be used to bridge routing instances.
4. **Answer: A,D**. The interfaces included in Trio tunnel services are gr-, ip-, lt-, mt-, pd-, pe-, ut-, and vt-.
5. **Answer: C**. Logical tunnels require that two IFLs that are to be joined use the `peer-unit` to reference the other. For example, lt-0/0/0.0 would use a `peer-unit` of 1, whereas lt-0/0/0.1 would use a `peer-unit` of 0.
6. **Answer: B**. Only two port mirroring instances can be associated to a single FPC.
7. **Answer: C**. The sampling formula is packets sampled = (`run-length` + 1) / `rate`. With a `rate` of 10 and `run-length` of 3, this would be packets sampled = (3 + 1) / 10. Otherwise written as 0.40 = 4 /10.

CHAPTER 8

Multi-Chassis Link Aggregation

IEEE 802.3ad is a great way to remove spanning tree from your network. However, IEEE 802.3ad doesn't work very well if one end of the bundle is split across two routers. Multi-Chassis Link Aggregation (MC-LAG) is a protocol that allows two routers to appear as single logical router to the other end of the IEEE 802.3ad bundle.

The most typical use case for MC-LAG in a Service Provider network is to provide customers both link-level and node-level redundancy. A good side effect of MC-LAG is that it removes the need for VPLS multi-homing (MH). For example, if a Service Provider had 4,000 VPLS instances that required node-level redundancy, one solution would be to implement VPLS MH; however, if there were a node failure, all 4,000 VPLS instances would have to signaled to move to the redundant PE router. The alternative is to use MC-LAG to provide node-level redundancy and eliminate 4,000 instances of VPLS MH; this method fails over the entire IFD in a single motion instead of every single VPLS MH instance.

Enterprise environments find that MC-LAG is a great method for multiple core routers to provide a single, logical IEEE 802.3ad interface to downstream switches and avoid having spanning tree block interfaces. From the perspective of a downstream switch the IEEE 802.3ad connection to the core is a single logical link, but in reality there are multiple core routers providing node-level redundancy.

Multi-Chassis Link Aggregation

MC-LAG allows a client device to establish IEEE 802.3ad across two physically separate chassis. A key differentiator is that MC-LAG maintains a separate control plane for each chassis that participates in the MC-LAG, as opposed to MX-VC where there also are two physical chassis, but the two control planes are virtualized into a single control plane.

Typically, when you setup IEEE 802.3ad it's only between two devices; the upside is that you now have link-level redundancy and more bandwidth, but the downside is that there isn't node-level redundancy. MC-LAG allows you to split the IEEE 802.3ad across two chassis to provide the node-level redundancy that's previously been missing when using vanilla IEEE 802.3ad. Let's take a look at a vanilla IEEE 802.3ad topology, as shown in Figure 8-1.

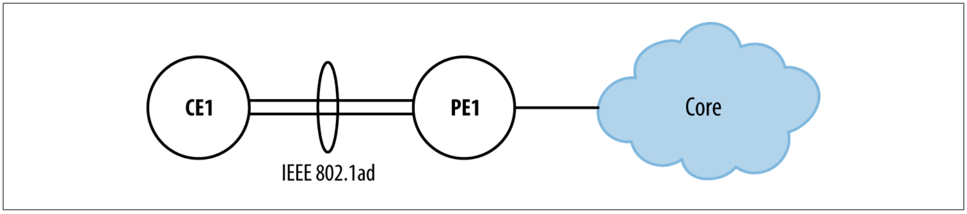


Figure 8-1. Vanilla IEEE 802.3ad

You can see that `CE1` is connected to `PE1` via IEEE 802.3ad, which contains two child links. The obvious benefit is that `CE1` now has twice the bandwidth because there are two child members and is able to survive a single link failure. If `PE1` were to fail, unfortunately that would leave `CE1` in the dark and unable to forward traffic to the core. What's needed is node-level redundancy on the provider side. The astute reader already realizes that vanilla IEEE 802.3ad will not work across multiple devices; that is where MC-LAG comes in.

If the provider were to install another router called `PE2`, this could provide the node-level redundancy that `CE1` is looking for. By running MC-LAG between `PE1` and `PE2`, the provider could provide link-level and node-level redundancy to `CE1` via IEEE 802.3ad. Let's take a look:

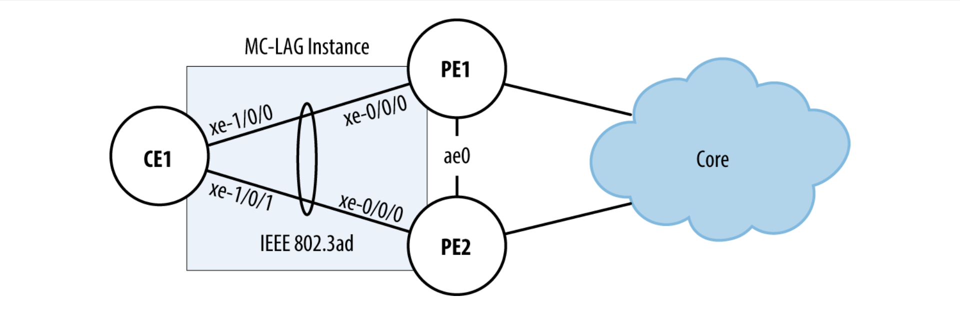


Figure 8-2. Simple Multi-Chassis Link Aggregation and IEEE 802.3ad.

Now the topology is becoming more interesting. Without getting into the details of MC-LAG, you can see that the router `CE1` has link-level redundancy via `xe-1/0/0` and `xe-1/0/1`, but also has node-level redundancy via routers `PE1` and `PE2`. If the router `PE1` were to have a failure, `CE1` would be able to forward traffic to the core via `PE2`.

Two of the great benefits of MC-LAG is that it is transparent to `CE1` and it doesn't require spanning tree. All of the configuration for MC-LAG is on the provider side on routers PE1 and PE2. The customer simply configures vanilla IEEE 802.3ad and isn't aware that there are actually two physical routers on the other side. Speaking of redundancy, let's go ahead and add node-level and link-level redundancy on the customer side with a second router CE2, as shown in Figure 8-3.

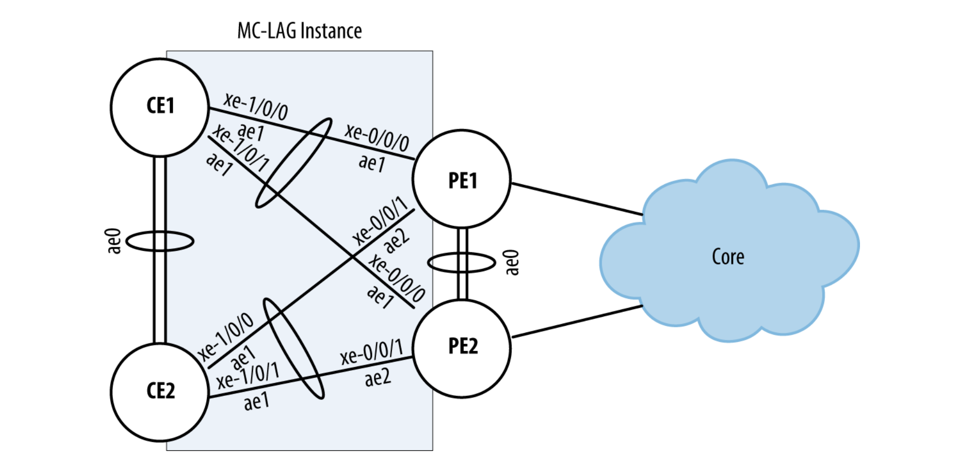


Figure 8-3. Full Node-Level and Link-Level Redundancy with MC-LAG and IEEE 802.3ad.

Now this is getting somewhere. From the perspective of the customer and provider, there's no single point of failure. For example, if CE1 fails, the router CE2 could take cover. On the flip side, if the provider router PE1 failed, the router PE2 will take cover and continue forwarding traffic. A vanilla IEEE 802.3ad was added between CE1 and CE2 as well as PE1 and PE2 to provide link-level redundancy between routers.

MC-LAG State Overview

When implementing MC-LAG into your network, one area of consideration is to decide which MC-LAG state to operate in. At a high level, MC-LAG is able to operate in Active-Standby or Active-Active state. When using the Active-Standby state, a nice benefit is that traffic forwarding is deterministic; the drawback is that a CE can only use half of the available links at any given time. The tradeoff with Active-Active is that the CE can take advantage of the full bandwidth of all available links, but traffic forwarding is nondeterministic.

MC-LAG Active-Standby

The MX-LAG Active-Standby state is similar to every other "active/[passive|standby|backup]" protocol in networking. When operating in Active-Standby state, only one of the routers in a MC-LAG redundancy group will be handling the data plane traffic from the downstream CE; this is elegantly handled by changing the state of one of the child links in the IEEE 802.3ad bundle to "Attached." This forces the CE to forward traffic down only one of the links.

Most users implement MC-LAG in Active-Standby state because it's easier to manage and the traffic forwarding is deterministic.

MC-LAG Active-Active

The alternative is to operate the MC-LAG in an Active-Active state, in which all CE child links in the IEEE 802.3ad bundle are in a state of "Collecting distributing," which allows the CE to forward traffic down both links. When implementing MC-LAG in Active-Active state, there's an additional configuration item called an Inter-Chassis Data Link (ICL). The ICL link is used to forward traffic between the PE chassis. For example, if the CE was forwarding traffic to both `PE1` and `PE2` equally, but the final egress port was on `PE1` interface `xe-0/0/0`, any traffic that was forwarded to `PE2` would need to traverse the ICL link so that it could be forwarded out the interface `xe-0/0/0` on `PE1`.

MC-LAG State Summary

Finding the right MC-LAG state is a bit of a balancing act, as there are many factors that weigh into this decision. For example, are you building your network to operate at line rate under failure conditions, or are you trying to provide as much bandwidth as possible? MC-LAG state is covered in more detail later in the chapter, where questions like this will be discussed.

MC-LAG Family Support

As of Junos 11.4, MC-LAG only supports Layer 2 families. Layer 3 can be supported indirectly through a routed interface and Virtual Router Redundancy Protocol (VRRP) associated with a bridge domain.

Bridge
: The most common use case is bridging Ethernet frames from a CE device. When using the Enterprise-style CLI, `family bridge` must be used.

VPLS
: Family VPLS and bridge are nearly identical from the vantage point of the Juniper MX. The use cases, however, are different. VPLS is a VPN service that rides on top of MPLS that provides a virtualized private LAN service. The only restriction is that `family vpls` can only be used with MC-LAG in `active-standby`.

CCC
: Cross-Connect Circuits (CCCs) are used to provide transparent point-to-point connections. This can be in the form of interface to interface, LSP to LSP, interface

to LSP, or LSP to interface. It's a bit misleading placing CCC in the MC-LAG Family Support, because strictly speaking `family ccc` isn't supported. However, `encapsulation ethernet-ccc` and `encapsulation vlan-ccc` are supported with MC-LAG. This allows you to switch Ethernet frames between two interfaces without MAC learning and minimal next-hop processing.

Multi-Chassis Link Aggregation versus MX Virtual-Chassis

MC-LAG and MX Virtual-Chassis (MX-VC) may appear to be the same at a high level. They both allow you to span physically different chassis and provide IEEE 802.3ad services to a CE, but there are some important differences that you should take into consideration before designing your network.

Table 8-1. MC-LAG and MX-VC Comparison as of Junos 11.4.

Feature	MC-LAG	MX-VC
Number of Control Planes	2	1
Centralized Management	No	Yes
Maximum Chassis[a]	2	2
Feature Implementation	Nondisruptive	Disruptive
Transparent to CE	Yes	Yes
Require IEEE 802.3ad	Yes.	No
State Replication Protocol	ICCP	VCCP
Require Spanning Tree	No	No
Require Dual REs per Chassis	No	Yes[b]
FPC Support	DPC and Trio[c]	Trio only
Require Special Hardware	No	No
State Control Options	Active-Passive and Active-Active	Active-Active
ISSU	Supported per chassis	Roadmap
Scale	Full routing engine scale per chassis	Limited to single routing engine across all chassis

[a] There is plan for MC-LAG and MX-VC to support more than two chassis, but as of Junos 11.4 the maximum number of chassis in a MX-VC or MC-LAG is two.

[b] The requirement for dual REs per chassis is currently in place because the maximum number of chassis in a MX-VC is two. As future releases of Junos support more chassis in a MX-VC, this dual RE requirement per chassis could be removed.

[c] DPC line cards only support MC-LAG Active-Standby. The Trio line cards support both Active-Standby and Active-Active.

The largest differentiation between the two is that MC-LAG is a simple protocol that runs between two routers to provide vanilla IEEE 802.3ad services whereas MX-VC is a more robust protocol that virtualizes multiple chassis into a single instance and offers more services beyond vanilla IEEE 802.3ad. On one hand, MC-LAG requires that two chassis keep and maintain control plane state, whereas MX-VC this functionality is built in as there is a single control plane, thus a single state; this difference will manifest itself into feature velocity. MX-VC will be able to support new features more frequently because of the architecture of a single control plane. However, MC-LAG will require incremental steps to support new features, as the new features need to be integrated into the MC-LAG protocol because of the requirement to keep and maintain state between chassis.

MC-LAG would require that chassis be managed separately, whereas MX-VC would virtualize the two chassis into a single virtual chassis and be managed as a single device. Another aspect to consider is the implementation process of each protocol. The implementation of MC-LAG can be done without major change and doesn't require a reboot of the chassis. In comparison, MX-VC requires a more in-depth configuration and requires a reboot, which causes disruption to customers.

Both MC-LAG and MX-VC have different methods of handling scale that need to be considered when designing your network. MC-LAG benefits from having two control planes, thus a control plane per chassis. For example, with MC-LAG, *each* chassis could have 64,000 IFLs and an IPv4 RIB capacity of 27 million. Because MX-VC has a single control plane and two chassis, the scale would be reduced to roughly half. For example, the two chassis in a MX-VC would *share* 64,000 IFLs and an IPv4 RIB capacity of 27 million.

MC-LAG Summary

Although similar to MX-VC, MC-LAG provides some key advantages depending on the use case: no disruption in service during the implementation and the control plane per chassis, which offers more scale, is retained. Each MC-LAG chassis is its own router and control plane, and must be managed separately. Even though each chassis is managed separately, MC-LAG provides a transparent and cross-chassis IEEE 802.3ad interface to the client.

Even though MX-VC has redundancy features to provide high availability during a failure scenario, it could be argued that MC-LAG provides an additional level of redundancy as it isn't subject to fate sharing. An example would be that if a network operator misconfigured a feature on MX-VC, it could potentially impact all chassis in the MX-VC, whereas if the same misconfiguration was on a router running MC-LAG, it would only impact that particular router.

Inter-Chassis Control Protocol

The Inter-Chassis Control Protocol (ICCP) is a simple and lightweight protocol that rides on top of TCP/IP that's used to maintain state, trigger failover, and ensure the MC-LAG configuration matches between the two chassis:

MC-LAG Configuration
ICCP is able to check the following attributes to ensure that the MC-LAG configurations between chassis are sane: MC-LAG port priority, system ID, aggregator ID, and port ID offset. If a misconfiguration is detected, MC-LAG will not come up properly; this is very helpful in quickly identifying and isolating operational MC-LAG problems.

State Information
In order to provide a transparent IEEE 802.3ad interface to a downstream CE, there are runtime objects that need to be synchronized between the two separate chassis: IGMP and DHCP snooping specific to interfaces participating in MC-LAG, MAC addresses that are learned or installed between the different chassis, and MC-LAG interfaces and their operational status.

Status Information
If the MC-LAG is running in an Active-Standby state, ICCP will need to keep track of which chassis is currently active versus in standby state.

Change Request
As the topology changes during a network event such as an interface doing down, ICCP will need to react accordingly; an example change request would be changing the MC-LAG state of Active from `PE1` to `PE2`.

The ICCP protocol is required when creating a MC-LAG configuration. Keep in mind that ICCP is just a simple control protocol and doesn't actually forward traffic between chassis; if traffic needs to be forwarded between chassis, it will be done with revenue ports between the chassis.

ICCP Hierarchy

As you design your MC-LAG network, there are a couple of guidelines that need to be followed. ICCP and MC-LAG are constructed in a hierarchy that has to match between routers participating in MC-LAG, as shown in Figure 8-4.

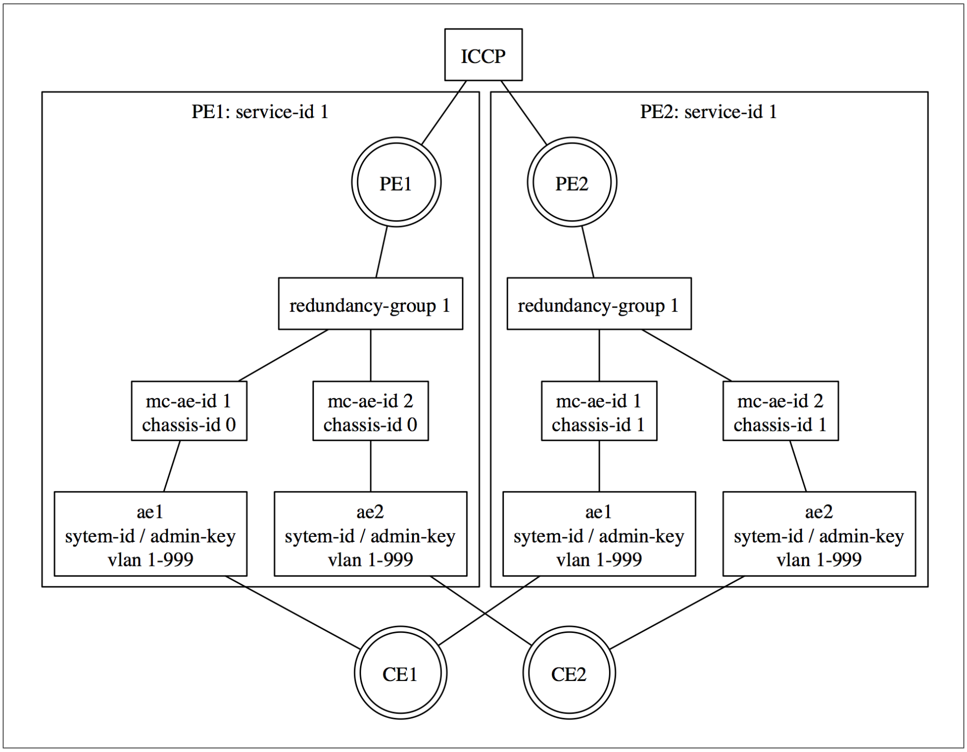


Figure 8-4. ICCP Hierarchy.

There are several major components that are used to construct the ICCP hierarchy: ICCP, PE routers, redundancy groups, multi-chassis aggregated Ethernet IDs, and aggregated Ethernet interfaces:

ICCP
The root of the hierarchy is ICCP. It's the control mechanism that provides a communication channel between PE routers.

Service Identifier
Because ICCP was designed to be scalable from the ground up, it was anticipated that ICCP may need to exist within a routing instance or logical system. The `service-id` is used to tie instances together across physical chassis. As of the writing of this book, MC-LAG is only supported in the default instance. Thus the `service-id` must match between chassis.

PE Routers
Routers that participate in MC-LAG need to establish a peering relationship with each other via ICCP.

Redundancy Groups

Redundancy groups are a collection of Multi-Chassis Aggregated Ethernet (MC-AE) IDs that share the same VLAN IDs. Redundancy groups act as a broadcast medium between PE routers so that application messages are concise and efficient. Notice that in Figure 8-4 `PE1` has a redundancy group ID of 1. This redundancy group contains MC-AE IDs 1 and 2, which share the same VLAN configuration of 1 through 999. If there was a new MAC address learned on `PE1` MC-AE ID 1, ICCP simply updates `PE2` redundancy group 1, instead of updating every single MC-AE ID with overlapping VLAN IDs. Each PE router is responsible for receiving ICCP updates, inspecting the redundancy group, then updating all MC-AE IDs that are part of the same redundancy group.

Multi-Chassis Aggregated Ethernet ID

MC-AE IDs are the operational glue between PE routers. In Figure 8-4, `CE1` is connected via IEEE 802.3ad to `PE1:ae1` and `PE1:ae2`. When configuring a logical MC-LAG interface that spans different PE routers, the MC-AE ID must match. When configuring multiple logical MC-LAG interfaces, you can use the `mc-ae-id` as a unique identifier to separate the logical interfaces. For example, all MC-LAG interfaces associated with `CE1` use `mc-ae-id 1`, whereas all MC-LAG interfaces associated `CE2` use `mc-ae-id 2`.

Chassis Identifier

The `chassis-id` is used by ICCP to uniquely identify each PE router when processing control packets. Each PE router that's configured for MC-LAG must have a unique chassis-id.

Aggregated Ethernet

Aggregated Ethernet (AE) interfaces are simply mapped on a 1:1 basis to MC-AE IDs. They follow the same peering rules as MC-AE IDs. The AE interfaces are where the actual client-facing configuration is constructed; this includes IEEE 802.3ad and Layer 2 configuration.

LACP System ID and Admin Key

Because two separate PE routers are being presented as a single logical router to the PE, two LACP attributes need to be synchronized across the PE routers: `system-id` and `admin-key`. The actual values aren't particularly important; all that matters is that they match across PE routers.

Status Control

This setting is used in the scenario where, if both of the PE routers boot up simultaneously, a particular router should become `active`. One chassis must be set to `active`, while the other chassis must be set to `standby`.

Keeping this simple hierarchy in mind when configuring MC-LAG will make life much easier. The design of ICCP allows for high-scale and complex topologies between PE and CE routers. Later in the chapter is a laboratory that will explore the depths of MC-LAG and show case the different topologies and ICCP control plane mechanisms.

ICCP Topology Guidelines

As of the writing of this book, the only supported topology for MC-LAG is between two PE routers. Although the ICCP protocol itself was designed to support additional scale, the support isn't there today. Figure 8-5 shows the MC-LAG topologies that are not supported.

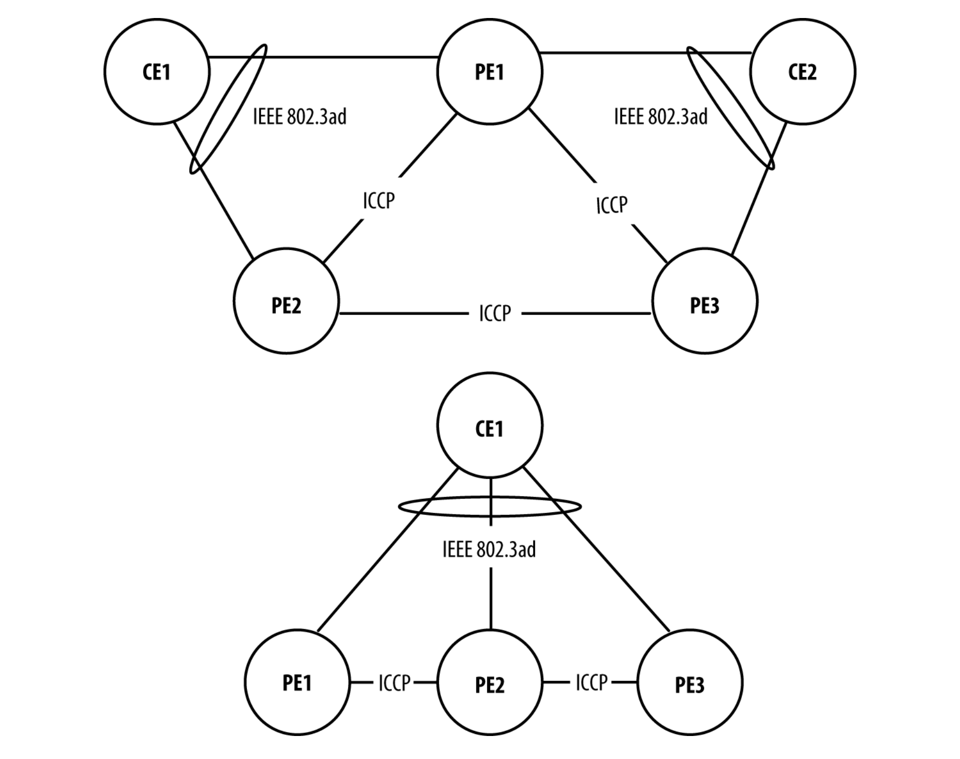


Figure 8-5. Unsupported MC-LAG Topologies.

Various forms of invalid MC-LAG topologies are shown in Figure 8-5. Each example has some form of three PE routers and a mixture of CE routers. The key point being that anything more than two PE routers will not be supported by MC-LAG, as of the writing of this book.

How to Configure ICCP

The configuration of ICCP is very straightforward. The only requirement is that the two routers have some sort of reachability, whether it's Layer 2 or Layer 3. The recommended method is to have both Layer 2 and Layer 3 reachability and use loopback or interface address peering. This method obviously requires an IEEE 802.1Q trunk between the two routers and some sort of IGP running between the routers advertising the loopbacks for reachability. Let's review a basic ICCP configuration example, as shown in Figure 8-6.

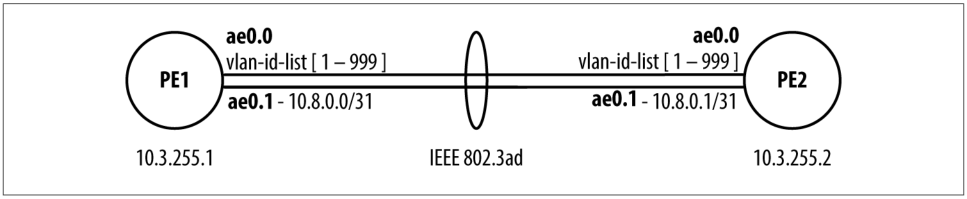


Figure 8-6. Vanilla ICCP Configuration

First things first. Let's make sure that the `service-id` is configured on both `PE1` and `PE2`. Recall that the `service-id` serves as a unique identifier for ICCP in the context of instances. In order to associate instances together, the `service-id` must match.

```
switch-options {
    service-id 1;
}
```

The `service-id` is set to 1 on both routers. This is an easy step to forget and leads to problems further down the road.

Routers `PE1` and `PE2` are connected via interface `ae0`, which has both Layer 2 and Layer 3 configurations. Let's review the configuration from the point of view of router `PE1`:

```
interfaces {
    ae0 {
        vlan-tagging;
        aggregated-ether-options {
            lacp {
                active;
                system-priority 100;
            }
        }
        unit 0 {
            family bridge {
                interface-mode trunk;
                vlan-id-list 1-999;
            }
        }
        unit 1 {
            vlan-id 1000;
            family inet {
                address 10.8.0.0/31;
            }
            family iso;
        }
    }
}
```

ICCP will be configured using the `ae0.1` addresses:

```
protocols {
    iccp {
        local-ip-addr 10.8.0.0;
        peer 10.8.0.1 {
            redundancy-group-id-list [ 1 2 ];
            liveness-detection {
                minimum-interval 150;
                multiplier 3;
            }
        }
    }
}
```

ICCP really only requires three arguments: local IP address, peer IP address, and which redundancy groups are in scope for this peer. In this example, the `local-ip-addr` is the interface address of PE1: `10.8.0.0`. The peer is the interface address of router PE2: `10.8.0.1`.

> This book has standardized on /31 addressing between point-to-point interfaces, but don't let the fancy /31 addressing fool you. `PE1` has an IP address of 10.8.0.0/31 and `PE2` has an IP address of 10.8.0.1/31. The `/31` addressing isn't required for ICCP at all. Recall that ICCP only needs some sort of TCP/IP reachability to the peer.

Because ICCP supports multiple peers, the configuration items are nested under each peer; this includes the redundancy group information and keepalive settings. The astute reader will recognize the `liveness-detection` as a setting for BFD. When designing the ICCP protocol, there was no need to reinvent the wheel, and using existing simple methods such as TCP/IP for transport and BFD for liveness detection make a lot of sense because they're well understood, easy to support, and work very well.

To help understand the redundancy group, let's take a look at the topology once more but add the CE, as shown in Figure 8-7.

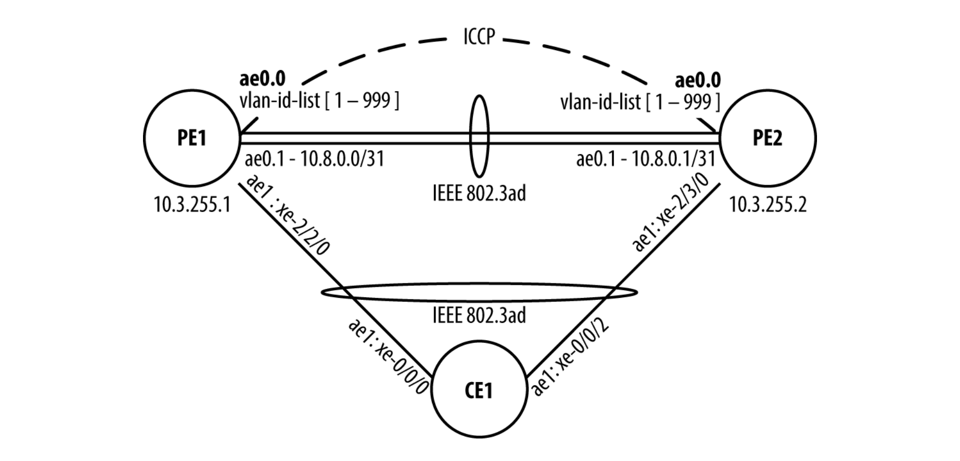


Figure 8-7. Adding CE1 to the Topology.

The ICCP protocol is illustrated with the dotted line going between routers `PE1` and `PE2`; recall that ICCP is using the routers' `ae0.1` interface for transport.

Now that ICCP is properly configured, let's verify that ICCP is up and operational.

```
{master}
dhanks@PE1-RE0>show iccp

Redundancy Group Information for peer 10.8.0.1
  TCP Connection       : Established
  Liveliness Detection : Up
  Redundancy Group ID          Status
    1                           Up

Client Application: lacpd
  Redundancy Group IDs Joined: 1

Client Application: l2ald_iccpd_client
  Redundancy Group IDs Joined: None

Client Application: MCSNOOPD
  Redundancy Group IDs Joined: None
```

The three most important items to look for are the TCP Connection, Liveliness Detection, and Redundancy Group Status. Recall that ICCP uses TCP/IP to transport the control packets. It's expected to see *Established* in the TCP Connection information. Any other status would indicate that there is a reachability problem between the two routers using the `local-ip-addr` and `peer` addresses given in the ICCP configuration. When using BFD liveliness detection with ICCP, the status will show up in the `show iccp` command as well. You can also verify this with `show bfd session detail`:

```
{master}
dhanks@PE1-RE0>show bfd session detail
```

```
                                                  Detect   Transmit
Address                  State     Interface      Time     Interval  Multiplier
10.8.0.1                 Up                       0.450     0.150        3
 Client ICCP realm 10.8.0.1, TX interval 0.150, RX interval 0.150
 Session up time 23:23:20
 Local diagnostic None, remote diagnostic None
 Remote state Up, version 1
 Replicated
                                                  Detect   Transmit
Address                  State     Interface      Time     Interval  Multiplier
10.8.0.1                 Up        ae0.1          0.450     0.150        3
 Client ISIS L2, TX interval 0.150, RX interval 0.150
 Session up time 1d 00:44, previous down time 00:00:04
 Local diagnostic NbrSignal, remote diagnostic AdminDown
 Remote state Up, version 1
 Replicated

2 sessions, 2 clients
Cumulative transmit rate 13.3 pps, cumulative receive rate 13.3 pps
```

It's interesting that `show bfd sessions` is indicating there are two sessions. Let's take a closer look with the `extensive` knob:

```
{master}
dhanks@PE1-RE0>show bfd session extensive
                                                  Detect   Transmit
Address                  State     Interface      Time     Interval  Multiplier
10.8.0.1                 Up                       0.450     0.150        3
 Client ICCP realm 10.8.0.1, TX interval 0.150, RX interval 0.150
 Session up time 23:23:28
 Local diagnostic None, remote diagnostic None
 Remote state Up, version 1
 Replicated
 Min async interval 0.150, min slow interval 1.000
 Adaptive async TX interval 0.150, RX interval 0.150
 Local min TX interval 0.150, minimum RX interval 0.150, multiplier 3
 Remote min TX interval 0.150, min RX interval 0.150, multiplier 3
 Local discriminator 10, remote discriminator 5
 Echo mode disabled/inactive
 Multi-hop route table 0, local-address 10.8.0.0

                                                  Detect   Transmit
Address                  State     Interface      Time     Interval  Multiplier
10.8.0.1                 Up        ae0.1          0.450     0.150        3
 Client ISIS L2, TX interval 0.150, RX interval 0.150
 Session up time 1d 00:44, previous down time 00:00:04
 Local diagnostic NbrSignal, remote diagnostic AdminDown
 Remote state Up, version 1
 Replicated
 Min async interval 0.150, min slow interval 1.000
 Adaptive async TX interval 0.150, RX interval 0.150
 Local min TX interval 0.150, minimum RX interval 0.150, multiplier 3
 Remote min TX interval 0.150, min RX interval 0.150, multiplier 3
 Local discriminator 6, remote discriminator 1
 Echo mode disabled/inactive
 Remote is control-plane independent
```

```
2 sessions, 2 clients
Cumulative transmit rate 13.3 pps, cumulative receive rate 13.3 pps
```

It now becomes clear as to why there are two BFD sessions. The ICCP client is configured for multi-hop, whereas the IS-IS client is configured as single-hop and is control plane independent. You can verify if BFD is running on the PFE or routing engine with the `show ppm` command:

```
{master}
dhanks@PE1-RE0>show ppm transmissions detail

Destination: 10.8.0.1, Protocol: BFD, Transmission interval: 150

Destination: 10.8.0.1, Protocol: BFD, Transmission interval: 150
Distributed, Distribution handle: 178, Distribution address: fpc2
```

Just as expected. The ICCP client is running on the routing engine while the IS-IS client is running on FPC2.

Let's use the `single-hop` knob under `liveness-detection` to change the ICCP client from multi-hop to single-hop. This will push the ICCP client down to FPC2 with the IS-IS client and reduce the load on the routing engine.

```
{master}[edit]
dhanks@PE1-RE0# set protocols iccp peer 10.8.0.1 liveness-detection single-hop

{master}[edit]
dhanks@R1-RE0# commit and-quit
re0:
configuration check succeeds
re1:
commit complete
re0:
commit complete
Exiting configuration mode
```

Don't forget to add the same configuration on PE2. Let's review the BFD sessions again and see if there is any change:

```
{master}
dhanks@PE1-RE0>show bfd session extensive
                                                  Detect   Transmit
Address                  State     Interface      Time     Interval  Multiplier
10.8.0.1                 Up        ae0.1          0.450     0.150        3
 Client ISIS L2, TX interval 0.150, RX interval 0.150
 Client ICCP realm 10.8.0.1, TX interval 0.150, RX interval 0.150
 Session up time 1d 00:46, previous down time 00:00:04
 Local diagnostic None, remote diagnostic None
 Remote state Up, version 1
 Replicated
 Min async interval 0.150, min slow interval 1.000
 Adaptive async TX interval 0.150, RX interval 0.150
 Local min TX interval 0.150, minimum RX interval 0.150, multiplier 3
 Remote min TX interval 0.150, min RX interval 0.150, multiplier 3
 Local discriminator 6, remote discriminator 1
 Echo mode disabled/inactive
 Remote is control-plane independent

1 sessions, 2 clients
Cumulative transmit rate 6.7 pps, cumulative receive rate 6.7 pps
```

Very cool; now both ICCP and IS-IS are clients of the same BFD session. Let's also verify that BFD has been pushed down to the PFE:

```
{master}
dhanks@PE1-RE0>show ppm transmissions detail

Destination: 10.8.0.1, Protocol: BFD, Transmission interval: 150
Distributed, Distribution handle: 178, Distribution address: fpc2
```

Perfect. BFD is now being handled by FPC2 and has relieved the routing engine from processing the BFD packets for both ICCP and IS-IS.

That was a nice detour with BFD, but let's get back on track. Recall that redundancy groups must match between PE routers; let's take a look at the configuration of the interfaces `xe-2/2/0` and `ae1` on router `PE1`:

```
interfaces {
    xe-2/2/0 {
        gigether-options {
            802.3ad ae1;
        }
    }
    ae1 {
        flexible-vlan-tagging;
        aggregated-ether-options {
            lacp {
                active;
                system-id 00:00:00:00:00:01;
                admin-key 1;
            }
            mc-ae {
                mc-ae-id 1;
                redundancy-group 1;
                chassis-id 0;
                mode active-standby;
                status-control active;
            }
        }
        unit 0 {
            family bridge {
                interface-mode trunk;
                vlan-id-list 100;
            }
        }
    }
}
```

The two routers PE1 and PE2 are comprised of the IEEE 802.3ad interface that's connected to the router CE1. On PE1, the interface xe-2/2/0 is a member of the aggregate interface ae1; on router PE2, the interface xe-2/3/0 is a member of the aggregate interface ae1. The glue that ties PE1:ae1 and PE2:ae1 together is the MC-AE ID, which in this example is mc-ae-id 1. Using this unique ID, the two routers PE1 and PE2 provide a common IEEE 802.3ad interface to CE1.

ICCP Configuration Guidelines

ICCP is designed using a strict hierarchy of objects that allow for high scale, flexibility, and future expansion of the protocol. As such, there are a few guidelines that need to be followed to ensure the proper configuration of ICCP.

- The service-id must match between the two PEs.
- The redundancy-group-id-list must match between the two PEs.

Any misconfiguration will result in ICCP or MC-LAG not operating properly. If you experience problems when configuring ICCP, be sure to check the service-id and redundancy-group-id-list as these two items must match between PE routers. It's easy to overlook, and time might be wasted troubleshooting other areas.

- Each PE router must have a unique chassis-id. This is used as a chassis identifier in the ICCP protocol.
- When assigning a mc-ae-id to an aggregated Ethernet interface, it must match on both PE routers so that the same mc-ae-id is presented to the CE.
- Although the same mc-ae-id is required on both PE routers, there's no requirement that the aggregated Ethernet match. For example, PE1 can have interface ae2 and PE2 can have interface ae3, but the mc-ae-id must be the same.
- When assigning the mc-ae-id to aggregated Ethernet interfaces on both routers, it must be part of the same redundancy-group.
- A single bridge-domain cannot correspond to two different redundancy groups. Recall that a redundancy group acts as a broadcast medium for a collection of MC-LAG interfaces. Thus a single bridge-domain can span multiple MC-LAG interfaces, but must be part of the same redundancy group.
- MC-LAG interfaces belonging to the same mc-ae-id need to have matching LACP system-id and admin-key.

Valid Configurations

Let's take a look at a couple of correct examples using the ICCP configuration guidelines.

Example 8-1. Vanilla MC-LAG Configuration.

```
PE1
ae1 {
    aggregated-ether-options {
        lacp {
            system-id 00:00:00:00:00:01;
            admin-key 1;
        }
        mc-ae {
            mc-ae-id 1;
            redundancy-group 1;
            chassis-id 0;
            mode active-active;
            status-control active;
        }
    }
    unit 0 {
        family bridge {
            interface-mode trunk;
            vlan-id-list 100;
        }
    }
}
PE2
ae1 {
    aggregated-ether-options {
        lacp {
            system-id 00:00:00:00:00:01;
            admin-key 1;
        }
        mc-ae {
            mc-ae-id 1;
            redundancy-group 1;
            chassis-id 1;
            mode active-active;
            status-control standby;
        }
    }
    unit 0 {
        family bridge {
            interface-mode trunk;
            vlan-id-list 100;
        }
    }
}
```

The most common configuration technique, as shown in Example 8-1, is to have matching aggregated Ethernet interface names between PE1 and PE2. This makes network operations much easier when having to troubleshoot an issue.

However, it isn't required that the aggregated Ethernet interface names match between the two PE routers. Let's take a look:

Example 8-2. Correct MC-LAG Configuration with Different Interface Names.

```
PE1
ae1 {
    aggregated-ether-options {
        lacp {
            system-id 00:00:00:00:00:01;
            admin-key 1;
        }
        mc-ae {
            mc-ae-id 1;
            redundancy-group 1;
            chassis-id 0;
            mode active-active;
            status-control active;
        }
    }
    unit 0 {
        family bridge {
            interface-mode trunk;
            vlan-id-list 100;
        }
    }
}
PE2
ae9 {
    aggregated-ether-options {
        lacp {
            system-id 00:00:00:00:00:01;
            admin-key 1;
        }
        mc-ae {
            mc-ae-id 1;
            redundancy-group 1;
            chassis-id 1;
            mode active-active;
            status-control standby;
        }
    }
    unit 0 {
        family bridge {
            interface-mode trunk;
            vlan-id-list 100;
        }
    }
}
```

There's no requirement that PE1 and PE2 have matching interface names. In Example 8-2, the router PE1 uses an interface name of ae1 while the router PE2 uses an interface name of ae9. This isn't recommended, but it's a valid configuration.

Invalid Configurations

Unfortunately, there are many ways to incorrectly configure MC-LAG, and they would be too numerous to list in this chapter. Let's review the most common mistakes:

Example 8-3. Invalid MC-LAG Configuration: chassis-id.

```
PE1
ae1 {
    aggregated-ether-options {
        lacp {
            system-id 00:00:00:00:00:01;
            admin-key 1;
        }
        mc-ae {
            mc-ae-id 1;
            redundancy-group 1;
            chassis-id 0;
            mode active-active;
            status-control active;
        }
    }
    unit 0 {
        family bridge {
            interface-mode trunk;
            vlan-id-list 100;
        }
    }
}
PE2
ae1 {
    aggregated-ether-options {
        lacp {
            system-id 00:00:00:00:00:01;
            admin-key 1;
        }
        mc-ae {
            mc-ae-id 1;
            redundancy-group 1;
            chassis-id 0;
            mode active-active;
            status-control standby;
        }
    }
    unit 0 {
        family bridge {
            interface-mode trunk;
            vlan-id-list 100;
        }
    }
}
```

The most common mistake is to use the same value for `chassis-id`. Because ICCP needs to uniquely identify each PE router, the `chassis-id` is required to be unique. Example 8-3 shows `PE1:ae1` and `PE2:ae1` with a `chassis-id` of 0. To correct this issue, `PE2:ae1chassis-id` needs to be changed to another value besides 0.

Let's move on to another example of an invalid configuration. Can you spot the problem?

Example 8-4. Invalid MC-LAG Configuration: mc-ae-id, redundancy-group, and status-control.

```
PE1
ae1 {
    aggregated-ether-options {
        lacp {
            system-id 00:00:00:00:00:01;
            admin-key 1;
        }
        mc-ae {
            mc-ae-id 1;
            redundancy-group 1;
            chassis-id 0;
            mode active-active;
            status-control active;
        }
    }
    unit 0 {
        family bridge {
            interface-mode trunk;
            vlan-id-list 100;
        }
    }
}
PE2
ae1 {
    aggregated-ether-options {
        lacp {
            system-id 00:00:00:00:00:01;
            admin-key 1;
        }
        mc-ae {
            mc-ae-id 2;
            redundancy-group 2;
            chassis-id 1;
            mode active-active;
            status-control active;
        }
    }
    unit 0 {
        family bridge {
            interface-mode trunk;
            vlan-id-list 100;
        }
    }
}
```

If you noticed that the `mc-ae-id` does not match between `PE1` and `PE2`, you would be correct. In Example 8-4, `PE1` has a `mc-ae-id` of 1 and `PE2` has a `mc-ae-id` of 2. These values need to match in order for the CE to successfully establish IEEE 802.3ad to PE1 and PE2. However, there are two other subtle issues that are causing MC-LAG problems.

Notice that PE1:ae1 and PE2:ae1 are both configured for IEEE 802.1Q for only VLAN ID 100. Recall that when two PE routers have a MC-LAG interface that share the same broadcast domain, it's required that the redundancy-group be the same.

The last problem in the configuration is that the status-control on both routers is set to active. This will cause ICCP to fail to negotiate the MC-LAG interfaces. Each router is mutually exclusive and has to be designated as either active or standby. To correct this problem, PE2:ae1 needs to change the status-control to standby.

ICCP Split Brain

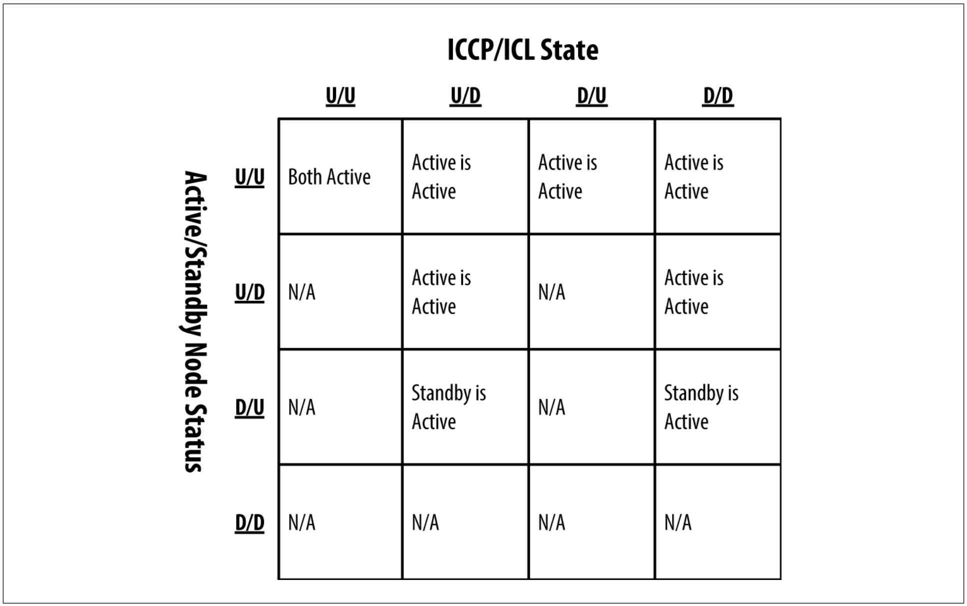
There are various kinds of failures—such as link and node failures—that could cause MC-LAG to change which PE is the active. Another failure scenario is that the communication of ICCP has failed, but each PE router is still operational and thinks its neighbor is down; this is referred to as a split brain scenario.

The first line of defense is to always be sure that ICCP peering is performed over loopback addresses. Loopback peering can survive link failures, assuming there's an alternate path between the two PE routers. The second line of defense is to explicitly define what happens when there's an ICCP failure.

The last line of defense is to explicitly configure how the two PE routers will behave in the event of a split brain failure. The goal is to deterministically identify the MC-LAG member that remains active in the event of a split brain. Each MCAE interface has an option to configure a feature called prefer-status-control-active. This option can only be configured on the MC-LAG member that is also configured for status-control active. The preferred MC-LAG member retains the configured LACP System ID while the other MC-LAG member falls back to its local LACP System ID.

Let's take the example that both PE routers are up and operational, but ICCP is down, and the result is a split brain scenario. The status-control active member will continue to use the configured LACP System ID on the MCAE interface that faces the CE. The other MC-LAG member will fall back and use its local LACP System ID. The result is that the CE receives different LACP System IDs; the CE will detach from the new peer who is sending the new LACP System ID and only forward traffic to the MC-LAG member that was configured with status-control active.

The matrix of information in Figure 8-8 describes which MC-LAG member remains active. U represents "Up" and D represents "Down."



Active/Standby Node Status \ ICCP/ICL State	U/U	U/D	D/U	D/D
U/U	Both Active	Active is Active	Active is Active	Active is Active
U/D	N/A	Active is Active	N/A	Active is Active
D/U	N/A	Standby is Active	N/A	Standby is Active
D/D	N/A	N/A	N/A	N/A

Figure 8-8. MC-LAG Prefer Status Control Matrix

For example, if both the active and standby MC-LAG members are up, but the ICCP is down and the ICL is up, the MC-LAG member configured as `status-control active` will remain as active.

ICCP Summary

ICCP is the glue that holds MC-LAG together. It's responsible for signaling changes within MC-LAG, updating state between the PE devices, and detecting MC-AE configuration issues. New users will find learning ICCP very easy, as it's based on simple protocol such as TCP/IP and BFD; this is evident by the three lines of configuration required to configure ICCP. Although ICCP can only support two PE routers as of Junos 11.4, the design of the protocol is so simple and extensible that it can easily allow for the addition of multiple PE devices in the future if required.

MC-LAG Modes

This chapter has touched on the two different MC-LAG modes: `active-standby` and `active-active`. When MC-LAG was first released, the only available mode was `active-standby`, which works on both DPC and MPC line cards. Because it was the first mode to be released and because of the simplicity of its design, the `active-standby` mode is generally more common. With the introduction of Trio and MPC line cards, MC-LAG was upgraded to support an `active-active` mode. This new mode will only work using Trio-based line cards such as the MPC.

Active-Standby

The `active-standby` mode works by selecting a PE router to be the active node while the other PE router is the standby node. Only one of the PE routers can be active at any given time. When a PE router is active, it will signal via LACP to the CE router its child link is available for forwarding.

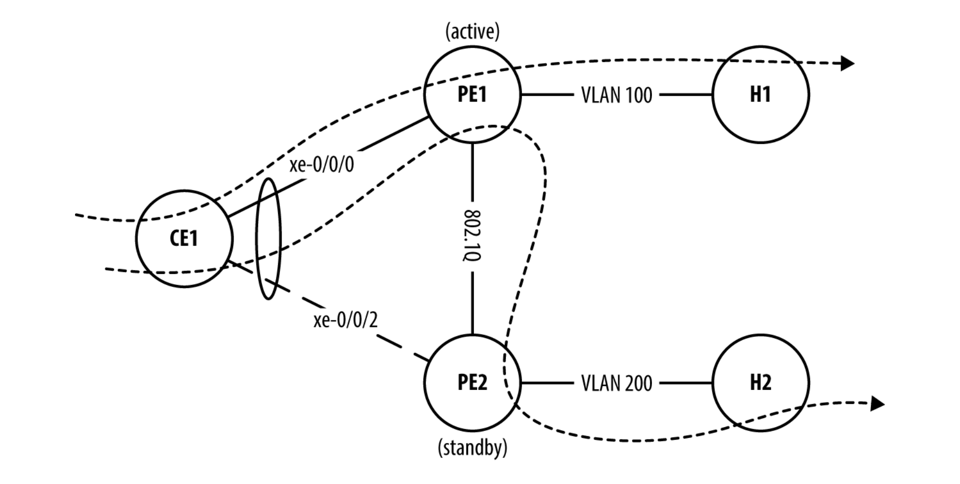


Figure 8-9. MC-LAG Mode Active-Standby.

Figure 8-9 illustrates MC-LAG in the `active-standby` mode. In this example, router `PE1` is active and `PE2` is the standby node. This mode forces all traffic through the active node `PE1`. For example, a frame destined to VLAN 100 would be forwarded to `PE1` and then directly to `H1`. A frame destined to VLAN 200 would also be forwarded to `PE1`, then to `PE2`, and finally to `H2`.

Let's take a look at the LACP information from the vantage point of `CE1`:

```
{master:0}
dhanks@CE1-RE0>show lacp interfaces
Aggregated interface: ae1
    LACP state:       Role   Exp   Def  Dist  Col  Syn  Aggr  Timeout  Activity
      xe-0/0/0       Actor    No    No   Yes  Yes  Yes   Yes     Fast    Active
      xe-0/0/0     Partner    No    No   Yes  Yes  Yes   Yes     Fast    Active
      xe-0/0/2       Actor    No    No    No   No  Yes   Yes     Fast    Active
      xe-0/0/2     Partner    No    No    No   No   No   Yes     Fast    Active
    LACP protocol:        Receive State  Transmit State          Mux State
      xe-0/0/0                  Current   Fast periodic Collecting distributing
      xe-0/0/2                  Current   Fast periodic           Attached
```

Notice how interface `xe-0/0/0` has a `Mux State` of `Collecting distributing` while the interface `xe-0/0/2` shows `Attached`. Such an elegant design for a simple concept; this method ensures that CE devices only need to speak LACP while the control packets and synchronization happen on the PE routers.

Let's take a look at the MC-LAG configuration on PE1:

```
ae1 {
    aggregated-ether-options {
        lacp {
            system-id 00:00:00:00:00:01;
            admin-key 1;
        }
        mc-ae {
            mc-ae-id 1;
            redundancy-group 1;
            chassis-id 0;
            mode active-standby;
            status-control active;
        }
    }
    unit 0 {
        family bridge {
            interface-mode trunk;
            vlan-id-list [ 100 200 ];
        }
    }
}
```

As expected, the router PE1 is configured in the **active-standby** mode and has been explicitly configured to by the active node. This can be verified through show commands as well:

```
{master}
dhanks@PE1-RE0>show interfaces mc-ae
 Member Link                   : ae1
 Current State Machine's State: mcae active state
 Local Status                  : active
 Local State                   : up
 Peer Status                   : standby
 Peer State                    : up
     Logical Interface         : ae1.0
     Topology Type             : bridge
     Local State               : up
     Peer State                : up
     Peer Ip/MCP/State         : N/A
```

The local state of PE1 shows active whereas the peer (PE2) shows as standby. Everything seems in order. Let's check PE2 as well:

```
{master}
dhanks@PE2-RE0>show interfaces mc-ae
 Member Link                   : ae1
 Current State Machine's State: mcae standby state
 Local Status                  : standby
 Local State                   : up
 Peer Status                   : active
 Peer State                    : up
     Logical Interface         : ae1.0
     Topology Type             : bridge
     Local State               : up
```

```
        Peer State              : up
        Peer Ip/MCP/State       : N/A
```

Just as expected. PE2 is showing the opposite of PE1. The local status is standby whereas the peer (PE1) is active.

Active-Active

The latest addition to the MC-LAG modes is active-active. This was introduced along with the Trio-based MPC line cards. As such, there is a restriction that MC-LAG operating in active-active mode must use MPC line cards; there's no support on older DPC line cards.

The active-active mode is very similar to active-standby with the exception that all child links on the CE device are active and can forward traffic to both PE routers. From the vantage point of the CE router, all child links will be in the Mux State of Collecting distributing. Figure 8-10 illustrates the possible traffic patterns given the two destinations of VLAN 100 and 200. Frames can be forwarded to either PE1 or PE2 and then to the final destination.

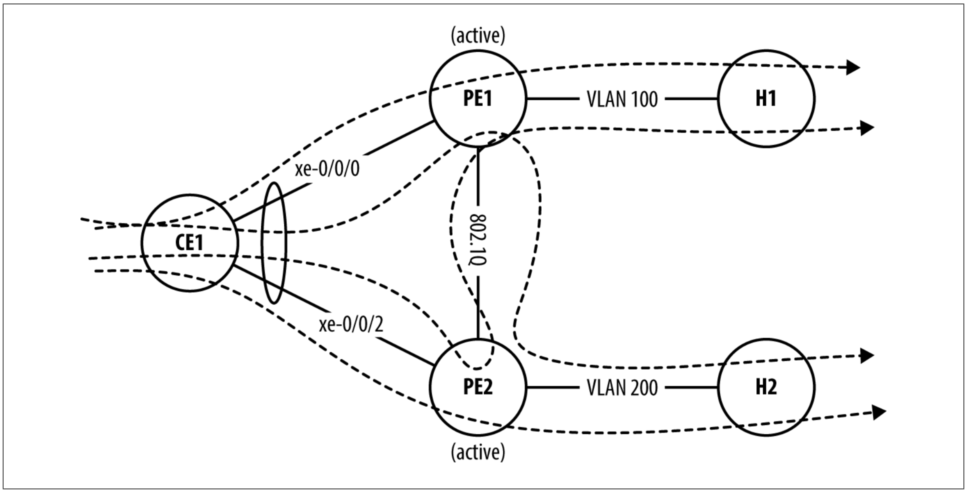


Figure 8-10. MC-LAG Active-Active

The active-active mode introduces a new component called Inter-Chassis Link (ICL). Although an ICL link isn't necessary with active-standby, it's required for active-active. The ICL link is simply an IEEE 802.1Q link between the two PE routers that are able to bridge all of the collective bridge domains on any interfaces participating in MC-LAG.

ICL Configuration

The configuration of the ICL link between the two PE routers is very simple. It's a standard IEEE 802.1Q IFL that contains all bridge domains that need to be protected. Figure 8-10 shows that VLANs 100 and 200 are being used. In this example, the ICL link on both PE1 and PE2 will need to include both VLANs 100 and 200:

```
ae0 {
    vlan-tagging;
    unit 0 {
        family bridge {
            interface-mode trunk;
            vlan-id-list [ 100 200 ];
        }
    }
}
```

The next step is to reference this newly defined ICL link within each MC-LAG interface. There are two methods to reference the ICL link: on the IFD or IFL of the MC-LAG interface.

Let's take a look at how to define the ICL protection at the IFD level on a MC-LAG interface on PE1:

```
interfaces {
    ae1 {
        flexible-vlan-tagging;
        multi-chassis-protection 10.8.0.1 {
            interface ae0;
        }
        encapsulation flexible-ethernet-services;
        aggregated-ether-options {
            lacp {
                system-id 00:00:00:00:00:01;
                admin-key 1;
            }
            mc-ae {
                mc-ae-id 1;
                redundancy-group 1;
                chassis-id 0;
                mode active-active;
                status-control active;
            }
        }
        unit 0 {
            family bridge {
                interface-mode trunk;
                vlan-id-list [ 100 200 ];
            }
        }
    }
}
```

The IFD method will provide ICL protection for the entire interface device of ae1. The only requirement is that the number of IFLs and the attributes must match. For example, the MC-LAG interface ae1.0 has VLANs 100 and 200, thus the ICL interface ae0.0 must have VLANs 100 and 200 as well. The IFD method serves as shortcut if the MC-LAG and ICL interfaces have the same number of IFLs, the same unit numbers, and the same VLAN definitions.

When defining the multi-chassis-protection, you must use the IP address of the ICCP peer. In this example, the peer is PE2 with an IP address of 10.8.0.1. The same is true for PE2; it must reference the ICCP address of PE1:

```
interfaces {
    ae1 {
        flexible-vlan-tagging;
        multi-chassis-protection 10.8.0.0 {
            interface ae0;
        }
        encapsulation flexible-ethernet-services;
        aggregated-ether-options {
            lacp {
                system-id 00:00:00:00:00:01;
                admin-key 1;
            }
            mc-ae {
                mc-ae-id 1;
                redundancy-group 1;
                chassis-id 0;
                mode active-active;
                status-control active;
            }
        }
        unit 0 {
            family bridge {
                interface-mode trunk;
                vlan-id-list [ 100 200 ];
            }
        }
    }
}
```

If it isn't possible to have matching MC-LAG and ICL interfaces, the alternative is to use a per IFL ICL protection. Let's take a look:

```
interfaces {
    ae1 {
        flexible-vlan-tagging;
        encapsulation flexible-ethernet-services;
        aggregated-ether-options {
            lacp {
                system-id 00:00:00:00:00:01;
                admin-key 1;
            }
            mc-ae {
                mc-ae-id 1;
                redundancy-group 1;
                chassis-id 0;
                mode active-active;
                status-control active;
            }
        }
        unit 99 {
        multi-chassis-protection 10.8.0.1 {
            interface ae0.0;
        }
            family bridge {
                interface-mode trunk;
                vlan-id-list [ 100 200 ];
            }
        }
    }
}
```

The only difference is that the `multi-chassis-protection` has moved from the IFD level into each IFL under the unit number. If the interface has multiple IFLs, the `multi-chassis-protection` must be defined for every IFL.

To verify that the ICL protection is up and running, use the `show interfaces mc-ae command` on router `PE1`:

```
{master}
dhanks@PE1-RE0>show interfaces mc-ae
 Member Link                  : ae1
 Current State Machine's State: mcae active state
 Local Status                 : active
 Local State                  : up
 Peer Status                  : active
 Peer State                   : up
     Logical Interface        : ae1.0
     Topology Type            : bridge
     Local State              : up
     Peer State               : up
     Peer Ip/MCP/State        : 10.8.0.1 ae0.0 up
```

The last section of the output indicates that the `multi-chassis-protection` is up and operational. Let's take a look at the same command but on router `PE2` instead:

```
{master}
dhanks@PE2-RE0>show interfaces mc-ae
 Member Link                  : ae1
 Current State Machine's State: mcae active state
 Local Status                 : active
 Local State                  : up
 Peer Status                  : active
 Peer State                   : up
     Logical Interface        : ae1.0
     Topology Type            : bridge
     Local State              : up
     Peer State               : up
     Peer Ip/MCP/State        : 10.8.0.0 ae0.0 up
```

Just as expected; the `multi-chassis-protection` is up and the only difference is the peer address, which from the vantage point of `PE2` is 10.8.0.0.

MAC Address Synchronization

When MC-LAG is operating in active-active mode, the CE is able to forward frames to both PE routers. This creates an interesting challenge with MAC learning. Figure 8-11 illustrates an example frame sourced from CE1 that's destined to H2. Let's assume that CE1 forwards the ARP request for H2 on xe-0/0/2 that's connected to PE2.

- PE2 broadcasts the ARP request for H2 out all interfaces that are associated with that bridge domain.
- H2 responds back to PE2 with an ARP reply.
- PE2 uses ICCP to install the H2 MAC address on PE1.
- PE2 forwards the ARP reply from H2 to CE1.

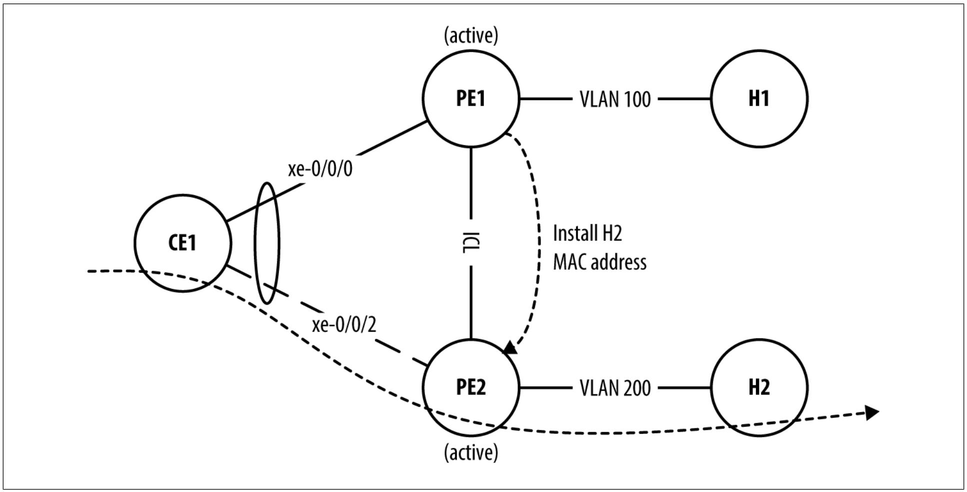


Figure 8-11. MC-LAG Active-Active MAC Address Synchronization.

Using ICCP to synchronize MAC addresses across both PE routers allows for an efficient flow of subsequent frames destined to H2. If CE1 forwarded a frame to xe-0/0/0 that was destined to H2, PE1 now has the MAC address of H2 installed and doesn't have to perform another ARP. PE1 can now forward any Ethernet frames to the ICL link that are destined to H2.

Chapter 2 showed you how to see the MAC address learning within broadcast domains. When using MC-LAG in active-active, it's possible to see how MAC addresses are installed remotely with show commands:

```
{master}
dhanks@PE1-RE0>show bridge mac-table

MAC flags (S -static MAC, D -dynamic MAC, L -locally learned
           SE -Statistics enabled, NM -Non configured MAC, R -Remote PE MAC)
```

```
Routing instance : default-switch
 Bridging domain : VLAN100, VLAN : 100
   MAC                 MAC      Logical
   address             flags    interface
   2c:6b:f5:38:de:c0   DR       ae2.0
   5c:5e:ab:6c:da:80   DL       ae1.0
```

Notice that each MAC address has a set of flags. Regarding MC-LAG `active-active`, the relevant flag is `Remote PE MAC` (`R`). In this example, the MAC address `2c:6b:f5:38:de:c0` was learned from `PE2` via ICCP. To get more information about this MAC address, let's use a different show command:

```
{master}
dhanks@PE1-RE0>show l2-learning redundancy-groups remote-macs

Redundancy Group ID : 1     Flags : Local Connect,Remote Connect

Service-id Peer-Addr VLAN MAC               MCAE-ID Subunit Opcode Flags Status
1          10.8.0.1  100  2c:6b:f5:38:de:c0 2       0        1     0     Installed
```

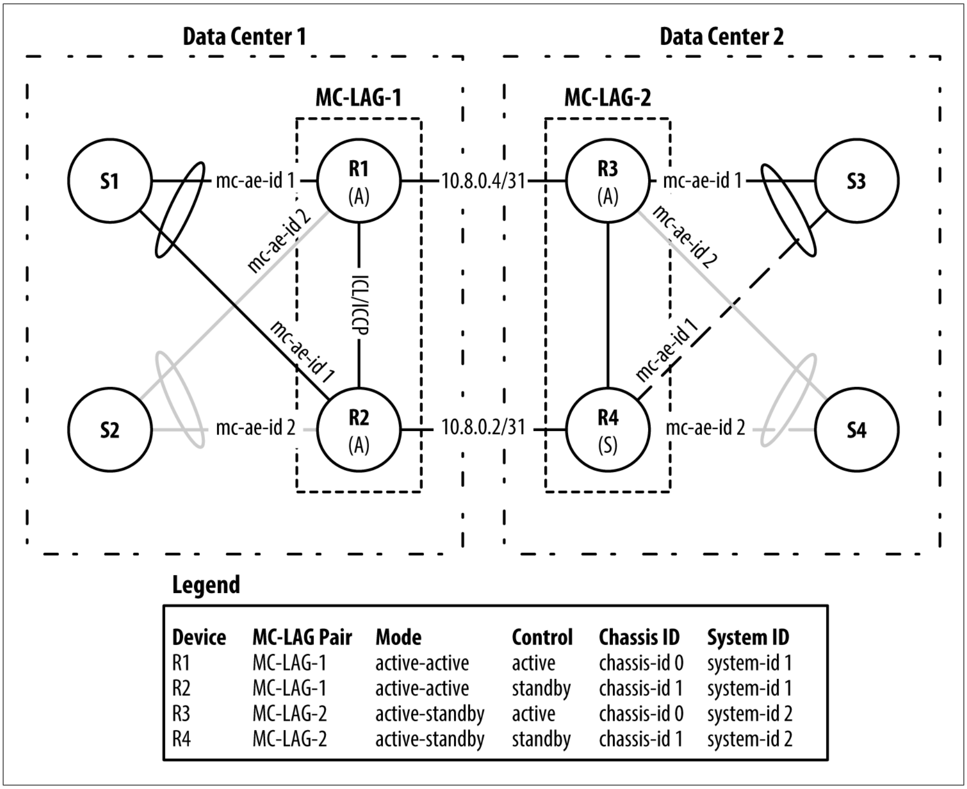
The `show l2-learning` command shows every detail from which the MAC address came. The MAC address `2c:6b:f5:38:de:c0` was learned from `PE2`, `service-id 1`, on VLAN 100. The `MCAE-ID` is the `mc-ae-id` on the remote PE router from which the MAC address was learned. If the MAC address was learned from an interface that doesn't participate in MC-LAG, the `MCAE-ID` will be omitted.

MC-LAG Modes Summary

MC-LAG can be configured to operate in an `active-active` or `active-standby` mode. Each mode has its own benefits and tradeoffs that have to be weighed carefully before being implemented in your network. The `active-active` mode allows you to fully utilize all available links and bandwidth, while at the same time providing high availability. The tradeoff is that it requires a higher degree of troubleshooting because of the MAC learning and the CE being able to hash traffic across both PE routers. The `active-standby` mode allows you to have deterministic traffic from the CE that's easy to troubleshoot. The tradeoff is that it will leave half of the available links in standby mode and unable to be utilized until there is a failure.

Case Study

The best way to apply the concepts in this chapter is to create a case study that integrates many of the MC-LAG features in a real-world scenario. Using the book's laboratory topology, it's possible to create a two pairs of PE routers and CE routers, as illustrated in Figure 8-12.



Device	MC-LAG Pair	Mode	Control	Chassis ID	System ID
R1	MC-LAG-1	active-active	active	chassis-id 0	system-id 1
R2	MC-LAG-1	active-active	standby	chassis-id 1	system-id 1
R3	MC-LAG-2	active-standby	active	chassis-id 0	system-id 2
R4	MC-LAG-2	active-standby	standby	chassis-id 1	system-id 2

Figure 8-12. MC-LAG Case Study Topology.

This case study will create two pairs of MC-LAG routers and two pairs of switches:

MC-LAG-1
: Routers `R1` and `R2` will be in the `active-active` mode. These are MX240 routers acting as the PE nodes.

MC-LAG-2
: Routers `R3` and `R4` will be in the `active-standby` mode. These are MX240 routers acting as the PE nodes.

Switch Pair 1
: Switches `S1` and `S2` will be running vanilla IEEE 802.3ad and IEEE 802.1Q. These are EX4500s acting as the CE nodes.

Switch Pair 2
: Switches `S3` and `S4` will be running vanilla IEEE 802.3ad and IEEE 802.1Q. These are EX4200s acting as the CE nodes.

On the far left and right are switches `S1` through `S4`. These switches are acting as vanilla CE devices connecting into their own MC-LAG instance. From the vantage point of each CE switch, it believes that it has a single IEEE 802.3ad connection going into the core of the topology. To mix things up, each MC-LAG instance will operate in a different mode. The MC-LAG instance for `S1` and `S2` will be `active-active`, whereas the MC-LAG instance for `S3` and `S4` will be `active-standby`.

This case study will move through all the different levels of the design starting with Layer 2 and working up all the way to higher level protocols such as ICCP. Once you have a full understanding of the design, the chapter will verify the design with show commands and provide commentary on what you see. Nearing the end of the case study, you will review several different verification scenarios to understand each step of MC-LAG and how the packet moves through each component.

Logical Interfaces and Loopback Addressing

To make the logical interface names easy to remember, the interface number matches the respective `mc-ae-id`. For example, on `R1` the MC-LAG instance going to `S1` uses `mc-ae-id 1`, thus the aggregated Ethernet interface on both `S1` and `R1` would be `ae1`.

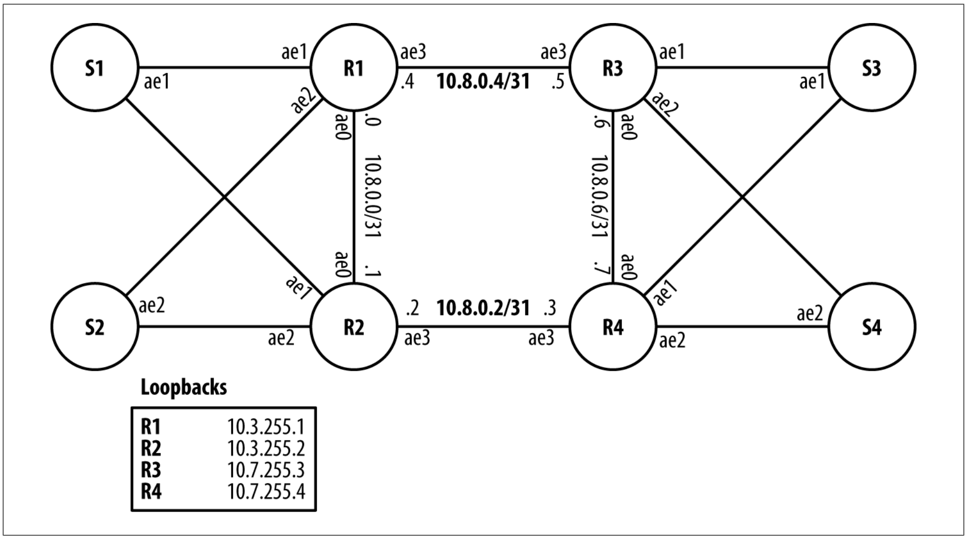


Figure 8-13. MC-LAG Case Study Logical Interfaces.

The astute reader will notice that some aggregated Ethernet interfaces contain two links whereas other interfaces contain only a single link. There are two scenarios in which the aggregated Ethernet interfaces contain only a single link:

MC-LAG Interfaces
: Although it isn't a requirement, this case study uses a single interface per router to construct a MC-LAG interface. For example, `R1` has a single interface in both `ae1` and `ae2`, while their complement is on `R2`. From the perspective of S1, the `ae1` aggregated interface has two links, each going to `R1` and `R2`.

Routed Interfaces

In an effort to make the interface topology less complex, the routed links that connect R1 to R3 and R2 to R4 are an aggregated interface. Interface ae3 is used to refer to the set of interfaces that connect the two sides of the topology together. So regardless of which vantage point is used, the interface ae3 will always refer to the router on the other side. For example, from the vantage point of R3, the interface ae3 will point toward R1. From the vantage point of R2, the interface ae3 will point toward R4.

Layer 2

There are two VLAN IDs per side with four VLAN IDs: 100, 200, 300, and 400. Each VLAN is associated with a particular network, as illustrated in Figure 8-14.

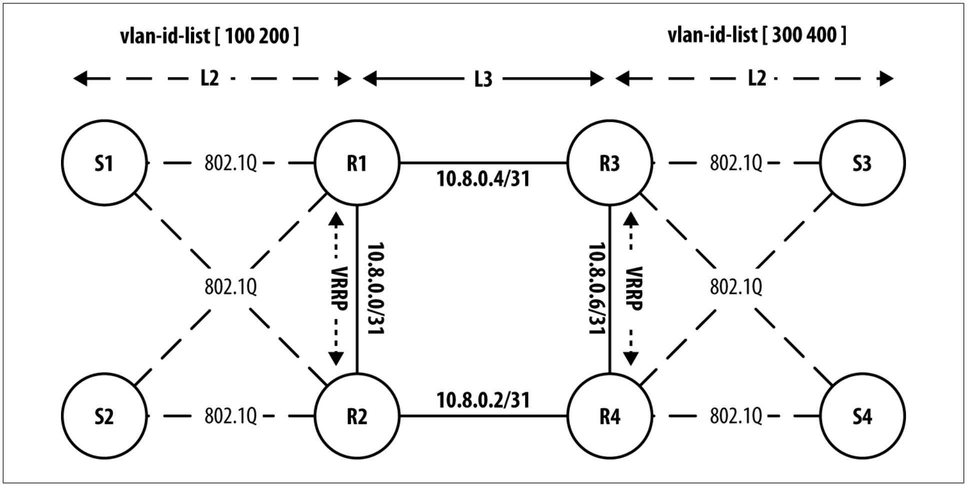


Figure 8-14. MC-LAG Case Study Layer 2 Topology.

The VLANs are split into two major groups: (S1, S2, R1, R2) and (R3, R4, S3, S4). Each group represents an island of Layer 2, which is common in multiple data center architecture.

Table 8-2. MC-LAG Case Study VLAN and IRB Assignments.

VLAN	Device	IRB
100	*VRRP*	192.0.2.1/26
100	R1	192.0.2.2/26
100	R2	192.0.2.3/26
100	S1	192.0.2.4/26
100	S2	192.0.2.5/26

VLAN	Device	IRB
200	*VRRP*	192.0.2.65/26
200	R1	192.0.2.66/26
200	R2	192.0.2.67/26
200	S1	192.0.2.68/26
200	S2	192.0.2.69/26
300	*VRRP*	192.0.2.129/26
300	R3	192.0.2.130/26
300	R4	192.0.2.131/26
300	S3	192.0.2.132/26
300	S4	192.0.2.133/26
400	*VRRP*	192.0.2.193/26
400	R3	192.0.2.194/26
400	R4	192.0.2.195/26
400	S3	192.0.2.196/26
400	S4	192.0.2.197/26

In summary, `S1`, `S2`, `R1`, and `R2` are assigned VLANs 100 and 200 while `R3`, `R4`, `S3`, and `S4` are assigned VLANs 300 and 400. Note that the VRRP addresses are running between (`R1`, `R2`) and (`R3`, `R4`). The VRRP addresses are used as the default gateway for downstream devices such as `S1` through `S4`.

The Layer 2 stops in the middle of the topology. `R1` to `R3` and `R2` to `R4` are separated by /31 routed interfaces. This effectively creates two islands of Layer 2 connectivity separated by two routed interfaces.

The interfaces between `R1` to `R2` and `R3` to `R4` have both `family inet` and `bridge` to support both Layer 2 and 3. It's common to combine Layer 2 and 3 on the same aggregate interface between core routers.

Loop Prevention

When using even the most basic MC-LAG configuration, there exists the physical possibility of a Layer 2 loop. For example, Figure 8-15 illustrates that from the perspective of a single instance of MC-LAG, the physical topology creates a triangle. It would seem logical that given this physical loop that some sort of loop prevention is required.

It's a common misconception that spanning tree is required when using MC-LAG because of the physical loop. However, MC-LAG has built-in loop prevention.

MC-LAG has loop prevention built into the protocol, thus traditional loop prevention protocols such as spanning tree aren't required.

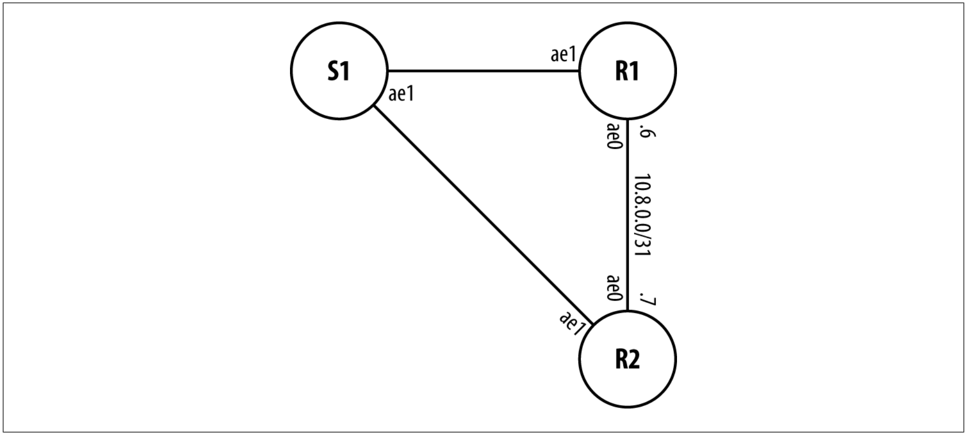


Figure 8-15. MC-LAG Case Study: Potential Layer 2 Loop.

MC-LAG places loop prevention in two places: ingress on the ICL link and egress on the MC-AE interfaces. The Trio chipset supports software features installed into the interfaces for both input and output. The feature for MC-LAG loop prevention is called `mclag-color` and `check-mclag-color`.

Input Feature. When using MC-LAG in an `active-active` mode, the ICL must apply ingress loop prevention. This case study has an `active-active` MC-LAG configuration between PE routers `R1` and `R2`. Let's review the ICL link configuration between `R1` and `R2`:

```
interfaces {
    ae0 {
        flexible-vlan-tagging;
        aggregated-ether-options {
            minimum-links 1;
            lacp {
                active;
                periodic fast;
            }
        }
        unit 0 {
            family bridge {
                interface-mode trunk;
                vlan-id-list [ 100 200 ];
            }
        }
        unit 1 {
            vlan-id 1000;
            family inet {
                address 10.8.0.0/31;
            }
            family iso;
        }
    }
}
```

It's important to note that the IFL `ae0.0` is used for the ICL as it can protect the VLAN IDs 100 and 200. The IFL `ae0.1` isn't bridged and acts as a routed IFL for Layer 3 connectivity between `R1` and `R2`.

In order to look at the `mclag-color` feature on the interface `ae0.0`, the use of PFE commands are required. The first step is to find the IFL index number for the interface `ae0.0`:

```
{master}
dhanks@R1-RE0>request pfe execute target fpc2 command "show interfaces" | match ae0
GOT:   128  ae0                  Ethernet     0x0000000000008000 local  Up
GOT:   324  ae0.0                VLAN Tagged    VPLS/Ethernet   0x000000002002c000
GOT:   325  ae0.1                VLAN Tagged    Ethernet        0x000000000000c000
GOT:   326  ae0.32767            VLAN Tagged    Ethernet        0x000000000400c000
```

The IFL index for the `ae0.0` interface is 324. Using this index number, you can look at the IFL input features to verify MC-LAG loop prevention:

```
{master}
dhanks@R1-RE0>request pfe execute target fpc2 command "show jnh if 324 input"
SENT: Ukern command: show jnh if 324 input
GOT:
GOT: ------- Input Features----------
GOT: Topology: ifl(324)
GOT:   Flavor: Input-IFL (49), Refcount 0, Flags 0x1
GOT:   Addr: 0x4ef91770, Next: 0x4e52c3e0, Context 0x144
GOT:     Link 0: b8a6cd41:c0000000, Offset 12, Next: 08a6cd60:00030000
GOT:     Link 1: b8a6cc81:c0000000, Offset 12, Next: 08a6cca0:00030000
GOT:     Link 2: 00000000:00000000, Offset 12, Next: 00000000:00000000
GOT:     Link 3: 00000000:00000000, Offset 12, Next: 00000000:00000000
GOT:
GOT: Topology Neighbors:
GOT:   [none]-> ifl(324)-> flist-master(iif)
GOT:         Feature List: iif
GOT:            [pfe-0]: 0x08a6cd6000030000;
GOT:            [pfe-1]: 0x08a6cca000030000;
GOT:           f_mask:0x08005100; c_mask:0xf0000000; f_num:24; c_num:4,  inst:-1
GOT:         Idx#4    set-iif:
GOT:            [pfe-0]: 0xa80003fffff00144
GOT:            [pfe-1]: 0xa80003fffff00144
GOT:
GOT:         Idx#17 mclag-color:
GOT:            [pfe-0]: 0x43687fffff800022
GOT:            [pfe-1]: 0x43687fffff800022
GOT:
GOT:         Idx#19  ptype-mux:
GOT:            [pfe-0]: 0xda000a6ca0000804
GOT:            [pfe-1]: 0xda000a6cbb800804
GOT:
GOT:         Idx#23 fabric-output:
GOT:            [pfe-0]: 0x2000000000000009
GOT:            [pfe-1]: 0x2000000000000009
```

```
35    GOT:
36    GOT: -------- Input Families --------
37    GOT:
38    GOT:         BRIDGE:
39    GOT:         Feature List: iff
40    GOT:            [pfe-0]: 0x0e011ef000020000;
41    GOT:            [pfe-1]: 0x0e017cf000020000;
42    GOT:           f_mask:0x00008000; c_mask:0x80000000; f_num:18; c_num:1,  inst:-1
43    GOT:         Idx#16 fwd-lookup:
44    GOT:            [pfe-0]: 0x0e011ef000020000
45    GOT:            [pfe-1]: 0x0e017cf000020000
46    GOT:
47    LOCAL: End of file
```

As shown on lines 24 through 26, the mclag-color feature is installed on index #17 in the feature list. This feature prevents any Ethernet frames from forming a loop over the ICL interface between R1 and R2.

Output Feature. Any type of MC-LAG MC-AE interfaces requires egress loop prevention. A similar process is used to view the check-mclag-color feature. In this case study, one of the MC-AE interfaces is ae1. Let’s review the MC-LAG configuration for this interface:

```
interfaces {
    xe-2/2/0 {
        gigether-options {
            802.3ad ae1;
        }
    }
    ae1 {
        flexible-vlan-tagging;
        multi-chassis-protection 10.8.0.1 {
            interface ae0;
        }
        aggregated-ether-options {
            lacp {
                active;
                periodic fast;
                system-id 00:00:00:00:00:01;
                admin-key 1;
            }
            mc-ae {
                mc-ae-id 1;
                redundancy-group 1;
                chassis-id 0;
                mode active-active;
                status-control active;
            }
        }
        unit 0 {
            family bridge {
                interface-mode trunk;
                vlan-id-list [ 100 200 ];
            }
        }
        }
    }
```

When viewing the egress MC-AE loop prevention feature, it's a similar process as viewing the ingress ICL feature. The exception is that the aggregated Ethernet interface cannot be used as an IFL index, but instead use the child interface. In this case, the child interface for `ae1` is `xe-2/2/0`. Let's determine the IFL index for `xe-2/2/0`:

```
{master}
dhanks@R1-
RE0>request pfe execute target fpc2 command "show interfaces" | match xe-2/2/0
GOT:   152  xe-2/2/0             Ethernet     0x0000000000008000     2  Up
GOT:   347  xe-2/2/0.0           VLAN Tagged    VPLS/Ethernet   0x000000002002c000
GOT:   346  xe-2/2/0.32767       VLAN Tagged    Ethernet        0x000000000400c000
```

In this case, the IFL index needed to view the `check-mclag-color` is 357. The same show command can be used, but this time the `output` option needs to be used:

```
{master}
dhanks@R1-RE0>request pfe execute target fpc2 command "show jnh if 347 output"
SENT: Ukern command: show jnh if 347 output
GOT:
GOT: -------- Output Features ---------
GOT: Topology: ifl(347)
GOT:   Flavor: Output-IFL (50), Refcount 2, Flags 0x1
GOT:   Addr: 0x4eec1050, Next: 0x4e871450, Context 0x15b
GOT:     Link 0: 00000000:00000000, Offset 12, Next: 00000000:00000000
GOT:     Link 1: 08a8e180:00030000, Offset 12, Next: 08a8e180:00030000
GOT:     Link 2: 00000000:00000000, Offset 12, Next: 00000000:00000000
GOT:     Link 3: 00000000:00000000, Offset 12, Next: 00000000:00000000
GOT:
GOT: Topology Neighbors:
GOT:   flist(IFBD EDMEM)-child(1)-> ifl(347)-> flist-master(oif)
GOT:   flist(IFBD EDMEM)-child(1)-+
GOT:         Feature List: oif
GOT:            [pfe-1]: 0x08a8e18000030000;
GOT:           f_mask:0x00a02080; c_mask:0xf0000000; f_num:26; c_num:4,  inst:1
GOT:         Idx#8    set-oif:
GOT:            [pfe-1]: 0x12e000200056ffff
GOT:
GOT:         Idx#10  ptype-mux:
GOT:            [pfe-1]: 0xda000a8dfb800804
GOT:
GOT:         Idx#18 check-mclag-color:
GOT:            [pfe-1]: 0x4b680040d3c00022
GOT:
GOT:         Idx#24 wan-output:
GOT:            [pfe-1]: 0x2400286000000000
GOT:
GOT: --------- Output Families --------
GOT:         BRIDGE:
GOT:         Feature List: off
GOT:            [pfe-1]: 0x08a6d71000010000;
GOT:           f_mask:0x80800000; c_mask:0xc0000000; f_num:11; c_num:2,  inst:1
GOT:         Idx#0 set-ifl-state:
```

```
38    GOT:            [pfe-1]: 0x12e000200056c5f2
39    GOT:
40    GOT:         Idx#8 redirect-check:
41    GOT:            [pfe-1]: 0x27fffff80000000c
42    GOT:
43    LOCAL: End of file
```

Lines 26 and 27 illustrate that the `check-mclag-color` feature is installed in the feature list at index #18. This specific feature prevents Ethernet loops that would be destined toward the CE.

Loop Prevention Verification. It's a good idea to see if the MC-LAG loop prevention features are installed, but actually seeing a counter of discarded packets is even better. There's an exception table stored in the PFE that contains a list of counters:

```
dhanks@R1-RE0>request pfe execute target fpc2 command "show jnh 0 exceptions"
SENT: Ukern command: show jnh 0 exceptions
GOT:
GOT: Reason                             Type      Packets      Bytes
GOT: ===============================================================
GOT:
GOT: PFE State Invalid
GOT: ----------------------
GOT: sw error                           DISC(64)        0          0
GOT: child ifl nonlocal to pfe          DISC(85)        0          0
GOT: invalid fabric token               DISC(75)        0          0
GOT: unknown family                     DISC(73)     6363     699302
GOT: unknown vrf                        DISC(77)        0          0
GOT: iif down                           DISC(87)       23       2596
GOT: unknown iif                        DISC( 1)
GOT: invalid stream                     DISC(72)        0          0
GOT: egress pfe unspecified             DISC(19)        0          0
GOT: invalid L2 token                   DISC(86)        0          0
GOT: mc lag color                       DISC(88)    79608    4620268
GOT: dest interface non-local to pfe    DISC(27)        0          0
GOT: invalid inline-svcs state          DISC(90)        0          0
GOT: nh id out of range                 DISC(93)        0          0
GOT: invalid encap                      DISC(96)        0          0
```

Throughout the life of this case study, the MC-LAG loop prevention has detected and discarded 79,608 packets.

It's unfortunate that PFE commands have to be used to see these MC-LAG loop prevention features and data. Until a CLI method of viewing this data becomes available, there's no other option.

R1 and R2

The PE routers `R1` and `R2` will host the VLANs 100 and 200 as well as the integrated routing and bridging interfaces. The great thing about MC-LAG is that it doesn't require the spanning tree protocol (STP). The two PE routers act as a single logical router, so in essence there's a single logical connection from the CE to the PE.

```
    bridge-domains {
        VLAN100 {
            vlan-id 100;
            routing-interface irb.100;
        }
        VLAN200 {
            vlan-id 200;
            routing-interface irb.200;
        }
    }
    interfaces {
        irb {
            unit 100 {
                family inet {
                    address 192.0.2.2/26 {
                        vrrp-group 0 {
                            virtual-address 192.0.2.1;
                            priority 101;
                            preempt;
                            accept-data;
                        }
                    }
                }
            }
            unit 200 {
                family inet {
                    address 192.0.2.66/26 {
                        vrrp-group 1 {
                            virtual-address 192.0.2.65;
                            priority 10;
                            accept-data;
                        }
                    }
                }
            }
        }
    }
```

Two very basic bridge domains are defined on R1 and R2 for VLAN 100 and 200; each VLAN has its respective irb interface. Two IFLs are defined on the irb interface and define the VRRP addresses, as illustrated in Table 8-2. The subtle difference is that VLAN 100 is master on R1 and VLAN 200 is master on R2; this allows the traffic to be load balanced between the two PE routers.

Bridging and IEEE 802.1Q. Both R1 and R2 have three aggregatedPAGE RANGE W/ 2 SUBs: Ethernet interfaces that participate in both bridging and IEEE 802.1Q: ae0, ae1, and ae2. The first IFL on each of the interfaces is configured identically to support family bridge and vlan-id-list [100 200]:

```
    interfaces {
        ae0 {
            flexible-vlan-tagging;
            aggregated-ether-options {
                minimum-links 1;
                lacp {
                    active;
                    periodic fast;
                }
            }
            unit 0 {
                family bridge {
                    interface-mode trunk;
                    vlan-id-list [ 100 200 ];
                }
            }
            unit 1 {
                vlan-id 1000;
                family inet {
                    address 10.8.0.0/31;
                }
                family iso;
            }
        }
        ae1 {
            flexible-vlan-tagging;
            multi-chassis-protection 10.8.0.1 {
                interface ae0;
            }
            aggregated-ether-options {
                lacp {
                    active;
                    periodic fast;
                    system-id 00:00:00:00:00:01;
                    admin-key 1;
                }
                mc-ae {
                    mc-ae-id 1;
                    redundancy-group 1;
                    chassis-id 0;
                    mode active-active;
                    status-control active;
                }
            }
            unit 0 {
                family bridge {
                    interface-mode trunk;
                    vlan-id-list [ 100 200 ];
                }
            }
        }
        ae2 {
            flexible-vlan-tagging;
            multi-chassis-protection 10.8.0.1 {
                interface ae0;
            }
            aggregated-ether-options {
                lacp {
                    active;
                    periodic fast;
                    system-id 00:00:00:00:00:01;
                    admin-key 1;
                }
                mc-ae {
                    mc-ae-id 2;
                    redundancy-group 1;
                    chassis-id 0;
                    mode active-active;
                    status-control active;
                }
            }
            unit 0 {
                family bridge {
                    interface-mode trunk;
                    vlan-id-list [ 100 200 ];
                }
            }
        }
    }
```

To verify that the Enterprise-style bridging configuration has successfully placed each interface into the appropriate bridge domain, the show command must be used:

```
{master}
dhanks@R1-RE0>show bridge domain

Routing instance        Bridge domain            VLAN ID     Interfaces
default-switch          VLAN100                  100         ae0.0
                                                             ae1.0
                                                             ae2.0
default-switch          VLAN200                  200         ae0.0
                                                             ae1.0
                                                             ae2.0
```

R1 has successfully found each of the three aggregated Ethernet interface and placed them both into bridge-domains VLAN100 and VLAN200.

IEEE 802.3ad. R1 and R2 have several interfaces that are part of IEEE 802.3ad. Recall the aggregated Ethernet interface naming convention, as shown in Table 8-3: the number of the aggregated Ethernet interface refers to the mc-ae-id of the CE. The interface ae3 always refers to the other PE router on the other side of the data center, and interface ae0 always connects the two PE routers within the same data center together.

Table 8-3. MC-LAG Case Study Aggregated Ethernet Matrix.

Device	Interface	Connected To
R1	ae0	R2
R2	ae0	R1
R1	ae1	S1
R2	ae1	S1
R1	ae2	S2

Device	Interface	Connected To
R2	ae2	S2
R1	ae3	R3
R2	ae3	R4

With a single command, it's possible to view the aggregated Ethernet interfaces, the LACP status, and child interfaces:

```
{master}
dhanks@R1-RE0>show lacp interfaces
Aggregated interface: ae0
    LACP state:       Role   Exp   Def  Dist  Col  Syn  Aggr  Timeout  Activity
      xe-2/0/0       Actor    No    No   Yes  Yes  Yes   Yes     Fast    Active
      xe-2/0/0     Partner    No    No   Yes  Yes  Yes   Yes     Fast    Active
      xe-2/0/1       Actor    No    No   Yes  Yes  Yes   Yes     Fast    Active
      xe-2/0/1     Partner    No    No   Yes  Yes  Yes   Yes     Fast    Active
    LACP protocol:        Receive State  Transmit State          Mux State
      xe-2/0/0                  Current   Fast periodic Collecting distributing
      xe-2/0/1                  Current   Fast periodic Collecting distributing

Aggregated interface: ae1
    LACP state:       Role   Exp   Def  Dist  Col  Syn  Aggr  Timeout  Activity
      xe-2/2/0       Actor    No    No   Yes  Yes  Yes   Yes     Fast    Active
      xe-2/2/0     Partner    No    No   Yes  Yes  Yes   Yes     Fast    Active
    LACP protocol:        Receive State  Transmit State          Mux State
      xe-2/2/0                  Current   Fast periodic Collecting distributing

Aggregated interface: ae2
    LACP state:       Role   Exp   Def  Dist  Col  Syn  Aggr  Timeout  Activity
      xe-2/3/0       Actor    No    No   Yes  Yes  Yes   Yes     Fast    Active
      xe-2/3/0     Partner    No    No   Yes  Yes  Yes   Yes     Fast    Active
    LACP protocol:        Receive State  Transmit State          Mux State
      xe-2/3/0                  Current   Fast periodic Collecting distributing

Aggregated interface: ae3
    LACP state:       Role   Exp   Def  Dist  Col  Syn  Aggr  Timeout  Activity
      xe-2/1/0       Actor    No    No   Yes  Yes  Yes   Yes     Fast    Active
      xe-2/1/0     Partner    No    No   Yes  Yes  Yes   Yes     Fast    Active
    LACP protocol:        Receive State  Transmit State          Mux State
      xe-2/1/0                  Current   Fast periodic Collecting distributing
```

The only aggregated Ethernet interface that has two members is the link between R1 and R2; interface ae0 is comprised of child interfaces xe-2/0/0 and xe-2/0/1. Interfaces ae1 and ae2 on R1 only have a single child interface going to their respective CE switches, because R2 contains the other redundant connection. For example, S1 has a single aggregated Ethernet interface ae0 that connects to both R1 and R2.

S1 and S2

As the switches S1 and S2 act as the CE devices, their configuration is much less complicated. From their vantage point, there's a single aggregated Ethernet interface that provides connectivity into the core of the network.

Bridging and IEEE 802.1Q. There are only two VLANs defined on S1 and S2: VLAN100 and VLAN200.

```
interfaces {
    ae1 {
        aggregated-ether-options {
            lacp {
                active;
                periodic fast;
            }
        }
        unit 0 {
            family ethernet-switching {
                port-mode trunk;
                vlan {
                    members all;
                }
            }
        }
    }
}
vlans {
    VLAN100 {
        vlan-id 100;
        l3-interface vlan.100;
    }
    VLAN200 {
        vlan-id 200;
        l3-interface vlan.200;
    }
}
```

Each VLAN has a Routed VLAN Interface (RVI)—which is the same thing as an IRB interface in MX-speak—defined to the vlan interface with its respective unit number that matches the VLAN ID.

```
interfaces {
    vlan {
        unit 100 {
            family inet {
                address 192.0.2.4/26;
            }
        }
    }
    vlan {
        unit 200 {
            family inet {
                address 192.0.2.68/26;
            }
            }
        }
    }
```

Each IFL has its own address in a /26 network that's associated with its respective VLAN ID. Later in the case study, these addresses will be used as part of the connectivity demonstration and failure scenarios.

Let's take a look at the VLANs to verify that the appropriate interfaces are associated with both VLAN IDs:

```
{master:0}
dhanks@S1-RE0>show vlans
Name           Tag     Interfaces
default
                       None
VLAN100        100
                       ae1.0*
VLAN200        200
                       ae1.0*
```

Just as expected, interface `ae1.0` is part of both VLANs and showing the proper VLAN ID.

The IEEE 802.1Q configurations for `S3` and `S4` are identical except for the VLAN definitions. In the case of `S3` and `S4`, VLAN 100 is replaced with VLAN 300 and VLAN 200 is replaced with VLAN 400.

IEEE 802.3ad. `S1` and `S2` have a single aggregated Ethernet interface that points into the core of the network; to be more specific, each of the child links `xe-0/0/0` and `xe-0/0/2` are connected to `R1` and `R2`.

```
interfaces {
    xe-0/0/0 {
        ether-options {
            802.3ad ae1;
        }
    }
    xe-0/0/2 {
        ether-options {
            802.3ad ae1;
        }
    }
    ae1 {
        aggregated-ether-options {
            lacp {
                active;
                periodic fast;
            }
        }
        unit 0 {
            family ethernet-switching {
                port-mode trunk;
                vlan {
                    members all;
                }
            }
        }
    }
}
```

From the vantage point of S1 and S2, there's just a single interface that connects to a single logical router. One of the largest strengths of MC-LAG is that it allows the CE to be happily unaware that it's connected to two PE routers and doesn't require any special configuration.

```
{master:0}
dhanks@S1-RE0>show lacp interfaces
Aggregated interface: ae1
    LACP state:       Role   Exp   Def  Dist  Col  Syn  Aggr  Timeout  Activity
      xe-0/0/0       Actor    No    No   Yes  Yes  Yes   Yes     Fast    Active
      xe-0/0/0     Partner    No    No   Yes  Yes  Yes   Yes     Fast    Active
      xe-0/0/2       Actor    No    No   Yes  Yes  Yes   Yes     Fast    Active
      xe-0/0/2     Partner    No    No   Yes  Yes  Yes   Yes     Fast    Active
    LACP protocol:        Receive State  Transmit State          Mux State
      xe-0/0/0                  Current   Fast periodic Collecting distributing
      xe-0/0/2                  Current   Fast periodic Collecting distributing
```

All packets on S1 entering or leaving the core are bridged over interface ae1. The interface xe-0/0/0 is connected to R1, whereas xe-0/0/2 is connected to R2. The Mux State of Collecting distributing on S1 indicates that both R1 and R2 are configured to be in a MC-LAG active-active state and are currently accepting traffic on both interfaces.

Layer 3

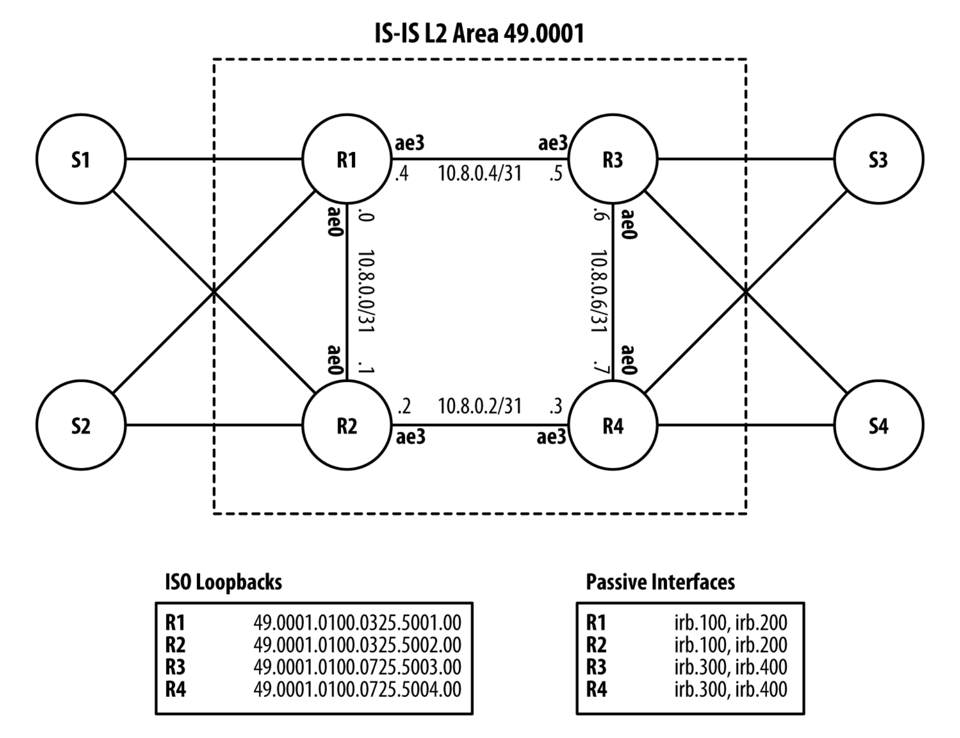
In this MC-LAG case study, various Layer 3 features are used to establish connectivity, distribute prefixes, and determine reachability. The PE nodes need to be able to provide gateway services to the CE switches, detect reachability errors, and provide cross data center connectivity. This section will cover IS-IS, VRRP, and BFD.

Interior Gateway Protocol—IS-IS

Each of the PE routers needs a method to advertise and distribute prefixes. This case study will use the IS-IS routing protocol. At a high level, all four of the PE routers will be part of a Level 2-only IS-IS area 49.0001. Let's review the IS-IS configuration of R1 and R2:

```
protocols {
    isis {
        reference-bandwidth 100g;
        level 1 disable;
        interface ae0.1 {
            point-to-point;
        }
        interface ae3.0 {
            point-to-point;
        }
        interface irb.100 {
            passive;
        }
        interface irb.200 {
            passive;
        }
        interface lo0.0 {
            passive;
        }
    }
}
```

The irb IFLs are included in the IS-IS configuration and set to passive. This method will include the IFAs into the link-state database (LSDB) but will not attempt to establish an adjacency over the interface. The IS-IS configuration for R3 and R4 are the same except interfaces irb.100 and irb.200 are replaced with irb.300 and irb.400.



ISO Loopbacks

R1	49.0001.0100.0325.5001.00
R2	49.0001.0100.0325.5002.00
R3	49.0001.0100.0725.5003.00
R4	49.0001.0100.0725.5004.00

Passive Interfaces

R1	irb.100, irb.200
R2	irb.100, irb.200
R3	irb.300, irb.400
R4	irb.300, irb.400

Figure 8-16. MC-LAG Case Study: IS-IS Area Design.

Given the IS-IS configuration of R1, there are two neighbors: R2 and R3. Let's verify with the show command:

```
{master}
dhanks@R1-RE0>show isis adjacency
Interface             System         L State        Hold (secs) SNPA
```

```
ae0.1                 R2-RE0         2  Up                     23
ae3.0                 R3             2  Up                     24
```

Each of the adjacencies is Up and operational. Let's take a look at the interfaces that are part of the IS-IS configuration:

```
{master}
dhanks@R1-RE0>show isis interface
IS-IS interface database:
Interface          L CirID Level 1 DR    Level 2 DR        L1/L2 Metric
ae0.1              2   0x1 Disabled      Point to Point          5/5
ae3.0              2   0x1 Disabled      Point to Point         10/10
irb.100            0   0x1 Passive       Passive               100/100
irb.200            0   0x1 Passive       Passive               100/100
lo0.0              0   0x1 Passive       Passive                 0/0
```

As expected, the interfaces ae0.1 and ae3.0 are participating in IS-IS as Level 2 only. There are also three interfaces defined as passive: irb.100, irb.200, and lo0.0. This will allow R1 to advertise the IRB and loopback addresses to its neighbors, but not attempt to establish an adjacency on these interfaces.

From the perspective of R1, there should be at least three other loopback addresses in the RIB: R2, R3, and R4.

```
dhanks@R1-RE0>show route protocols isis

inet.0: 22 destinations, 22 routes (22 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

10.8.0.6/31        *[IS-IS/18] 03:09:17, metric 20
> to 10.8.0.5 via ae3.0
10.3.255.2/32      *[IS-IS/18] 6d 06:06:52, metric 5
> to 10.8.0.1 via ae0.1
10.7.255.3/32      *[IS-IS/18] 03:09:17, metric 10
> to 10.8.0.5 via ae3.0
10.7.255.4/32      *[IS-IS/18] 6d 06:06:42, metric 15
> to 10.8.0.1 via ae0.1
10.8.0.2/31        *[IS-IS/18] 6d 06:06:52, metric 15
> to 10.8.0.1 via ae0.1
192.0.2.128/26     *[IS-IS/18] 02:53:55, metric 73
> to 10.8.0.5 via ae3.0
192.0.2.192/26     *[IS-IS/18] 02:53:55, metric 73
> to 10.8.0.5 via ae3.0
```

As expected, the three router loopbacks of R2, R3, and R4 are present. There are also additional IS-IS routes: two /31 and two /26 networks. Recall that in Figure 8-16, there are /31 networks connecting the PE routers together. The 10.8.0.6/31 ties together R3 to R4 and 10.8.0.2/31 ties together R2 to R4. The two /26 networks are the irb.300 and irb.400 interfaces on R3 and R4.

Bidirectional Forwarding Detection

BFD is a very simple echo protocol that is routing protocol independent that enables subsecond failover. One of the major benefits to using BFD is that multiple clients such as IS-IS and ICCP can use a single BFD session in order to detect forwarding errors. This eliminates having to set and manage multiple timers with different clients that may or may not support subsecond failure detection.

Using Junos apply-groups is an easy way to make sure that every aggregated Ethernet interface configured in protocols isis is configured to use BFD.

```
groups {
    bfd {
        protocols {
            isis {
                interface <ae*> {
                    bfd-liveness-detection {
                        minimum-interval 150;
                        multiplier 3;
                    }
                }
            }
        }
    }
}
apply-groups [ bfd ];
```

This apply-group will walk down into the protocols isis interface level and attempt to find any interfaces matching <ae*> and apply a generic BFD configuration to each interface match. The display inheritance option is used to display which interfaces were affected by the apply-group bfd:

```
{master}
dhanks@R1-RE0>show configuration protocols isis | display inheritance
reference-bandwidth 100g;
level 1 disable;
interface ae0.1 {
    point-to-point;
    ##
    ## 'bfd-liveness-detection' was inherited from group 'bfd'
    ##
    bfd-liveness-detection {
        ##
        ## '150' was inherited from group 'bfd'
        ##
        minimum-interval 150;
        ##
        ## '3' was inherited from group 'bfd'
        ##
        multiplier 3;
    }
}
interface ae3.0 {
    point-to-point;
    ##
    ## 'bfd-liveness-detection' was inherited from group 'bfd'
    ##
    bfd-liveness-detection {
        ##
```

```
            ## '150' was inherited from group 'bfd'
            ##
            minimum-interval 150;
            ##
            ## '3' was inherited from group 'bfd'
            ##
            multiplier 3;
        }
    }
    interface irb.100 {
        passive;
    }
    interface irb.200 {
        passive;
    }
    interface lo0.0 {
        passive;
    }
```

This example shows that interfaces `ae0.1` and `ae3.0` inherited the BFD configuration automatically because their interfaces names matched `<ae*>`. Let's also verify the BFD configuration with `show bfd sessions` to ensure connectivity to other neighbors:

```
{master}
dhanks@R1-RE0>show bfd session
                                              Detect   Transmit
Address          State      Interface         Time     Interval  Multiplier
10.8.0.1         Up         ae0.1             0.450     0.150        3
10.8.0.5         Up         ae3.0             0.450     0.150        3

2 sessions, 3 clients
Cumulative transmit rate 13.3 pps, cumulative receive rate 13.3 pps
```

The two sessions are to be expected: a session going to `R3` and another going to `R4`. The interesting thing to note is that there are three clients. Because IS-IS is the only BFD client we've configured so far, it's safe to assume that there should only be two clients. Where is the third client coming from? Let's use the `extensive` option to see more detail:

```
{master}
dhanks@R1-RE0>show bfd session extensive | no-more
                                              Detect   Transmit
Address          State      Interface         Time     Interval  Multiplier
10.8.0.1         Up         ae0.1             0.450     0.150        3
Client ICCP realm 10.8.0.1, TX interval 0.150, RX interval 0.150
Client ISIS L2, TX interval 0.150, RX interval 0.150
Session up time 6d 02:41, previous down time 00:04:17
Local diagnostic CtlExpire, remote diagnostic CtlExpire
 Remote state Up, version 1
 Replicated
 Min async interval 0.150, min slow interval 1.000
 Adaptive async TX interval 0.150, RX interval 0.150
 Local min TX interval 0.150, minimum RX interval 0.150, multiplier 3
 Remote min TX interval 0.150, min RX interval 0.150, multiplier 3
 Local discriminator 4, remote discriminator 6
 Echo mode disabled/inactive
```

```
18    Remote is control-plane independent
19
20                                           Detect   Transmit
21    Address        State     Interface      Time     Interval  Multiplier
22    10.8.0.5       Up        ae3.0           0.450     0.150        3
23    Client ISIS L2, TX interval 0.150, RX interval 0.150
24    Session up time 6d 02:40
25    Local diagnostic None, remote diagnostic None
26    Remote state Up, version 1
27    Replicated
28    Min async interval 0.150, min slow interval 1.000
29    Adaptive async TX interval 0.150, RX interval 0.150
30    Local min TX interval 0.150, minimum RX interval 0.150, multiplier 3
31    Remote min TX interval 0.150, min RX interval 0.150, multiplier 3
32    Local discriminator 8, remote discriminator 3
33    Echo mode disabled/inactive
34    Remote is control-plane independent
35
36    2 sessions, 3 clients
37    Cumulative transmit rate 13.3 pps, cumulative receive rate 13.3 pps
```

Aha! Line 6 indicates that the third client is ICCP. Recall previously in the chapter that ICCP uses BFD for failure detection and that BFD is able to support multiple clients per session. The session associated with interface ae0.1 has two clients: ICCP and ISIS. This makes sense because both ISIS and ICCP are configured between R1 and R2.

Virtual Router Redundancy Protocol

In order for R1 and R2 to provide consistent gateway services to S1 and S2, a common gateway address needs to be used that will survive a PE failure. The most common method to provide gateway services between routers is VRRP.

```
interfaces {
    irb {
        unit 100 {
            family inet {
                address 192.0.2.2/26 {
                    vrrp-group 0 {
                        virtual-address 192.0.2.1;
                        priority 101;
                        preempt;
                        accept-data;
                    }
                }
            }
        }
        unit 200 {
            family inet {
                address 192.0.2.66/26 {
                    vrrp-group 1 {
                        virtual-address 192.0.2.65;
                        priority 10;
                        accept-data;
                    }
                    }
                }
            }
        }
    }
```

The VRRP address for `VLAN100` is 192.0.2.1/26, and the VRRP address for `VLAN200` is 192.0.2.65/26. The configuration is very easy and only requires the addition of a `vrrp-group` followed by the `virtual-address` and an appropriate `priority`. In this MC-LAG case study, `R1` will be the VRRP master for 192.0.2.1 while `R2` will be the VRRP master for 192.0.2.65. Let's verify with the `show vrrp` command:

```
{master}
dhanks@R1-RE0>show vrrp
Interface  State  Group   VR state VR Mode  Timer    Type  Address
irb.100    up         0   master   Active      A  0.239 lcl  192.0.2.2
                                                       vip  192.0.2.1
irb.200    up         1   backup   Active      D  3.636 lcl  192.0.2.66
                                                       vip  192.0.2.65
                                                       mas  192.0.2.67
```

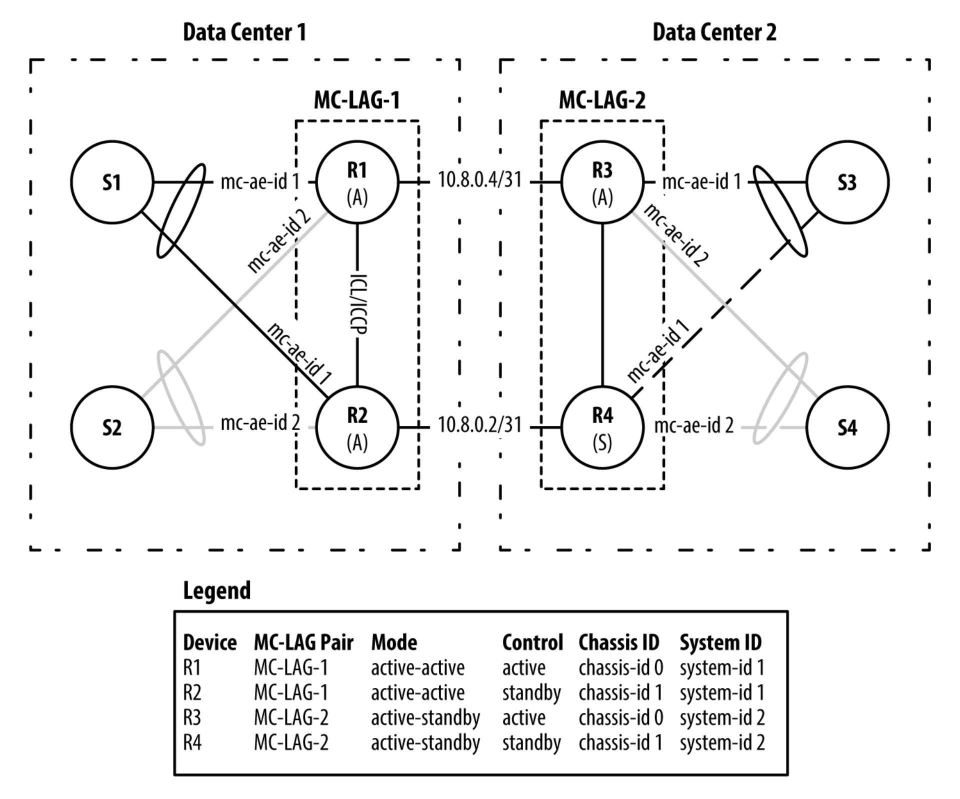
Each VRRP address is in an `up` state, and `R1` proves to be the VRRP master for 192.0.2.1 while `R2` is the VRRP master for 192.0.2.65. Let's verify connectivity from `S1` to the VRRP address 192.0.2.1:

```
{master:0}
dhanks@S1-RE0>ping source 192.0.2.4 192.0.2.1
PING 192.0.2.1 (192.0.2.1): 56 data bytes
64 bytes from 192.0.2.1: icmp_seq=0 ttl=62 time=1.972 ms
64 bytes from 192.0.2.1: icmp_seq=1 ttl=62 time=4.222 ms
64 bytes from 192.0.2.1: icmp_seq=2 ttl=62 time=4.113 ms
64 bytes from 192.0.2.1: icmp_seq=3 ttl=62 time=1.556 ms
64 bytes from 192.0.2.1: icmp_seq=4 ttl=62 time=1.549 ms
64 bytes from 192.0.2.1: icmp_seq=5 ttl=62 time=1.293 ms
```

Outstanding. Now that connectivity has been verified let's begin to review the higher-level protocols such as ICCP and MC-LAG.

MC-LAG Configuration

At a high level, there are two pairs of PE routers that participate in MC-LAG. Each PE pair has two MC-AE interfaces that correspond to each of the CE devices. In Figure 8-17, the first pair of PE routers are `R1` and `R2`; they have two MC-AE instances that correspond to `S1` and `S2`.



Device	MC-LAG Pair	Mode	Control	Chassis ID	System ID
R1	MC-LAG-1	active-active	active	chassis-id 0	system-id 1
R2	MC-LAG-1	active-active	standby	chassis-id 1	system-id 1
R3	MC-LAG-2	active-standby	active	chassis-id 0	system-id 2
R4	MC-LAG-2	active-standby	standby	chassis-id 1	system-id 2

Figure 8-17. MC-LAG Case Study: MC-LAG Topology.

The second pair of PE routers are R3 and R4, which contain two MC-AE instances that correspond to S3 and S4. The only difference between the two PE pairs is that R1 and R2 are configured to be `active-active`, whereas R3 and R4 are configured to be `active-standby`.

ICCP

The first step to building out the MC-LAG case study is to configure ICCP. There are two locations where ICCP needs to be installed: between R1 and R2 and between R3 and R4. As described previously, these are the two pairs of PE routers and will require state synchronization with ICCP to provide IEEE 802.3ad services to CE devices.

R1 and R2. Recall that ICCP rides on top of TCP/IP, a good method for establishing connectivity between R1 and R2 would be to use the 10.8.0.0/31 network. Let's review the ICCP configuration on R1 to learn more:

```
protocols {
    iccp {
        local-ip-addr 10.8.0.0;
        peer 10.8.0.1 {
            redundancy-group-id-list 1;
            liveness-detection {
                minimum-interval 150;
                multiplier 3;
                single-hop;
            }
            authentication-key "$9$dzw2ajHmFnCZUnCtuEhVwY"; ## SECRET-DATA
        }
    }
}
```

There are three major components that are required when configuring ICCP. This case study will use an additional two components to improve failure detection and security:

Local IP Address

The `local-ip-addr` is a required component. This is the IP address that is used to source the ICCP traffic. The IP address must be present on the local router such as an interface address or loopback address.

Peer IP Address

The `peer` is a required component. This is the destination IP address of the peer router. It's required that this IP address be present on the peer router such as an interface address or loopback address.

Redundancy Group ID List

The `redundancy-group-id-list` is a required component. Every `redundancy-group` used in the configuration of MC-AE interfaces must be installed into ICCP. This case study will use multiple MC-AE interfaces but only a single `redundancy-group`.

Liveness Detection

The `liveness-detection` is an optional component. This will invoke a BFD session to the peer router and install ICCP as a client. This example will use a `minimum-interval` of 150 and a `multiplier` of 3. These options will be able to detect a forwarding error in 450 ms. The hidden option `single-hop` will force BFD to not use multi-hop; this enables the distribution of BFD down to the line cards and away from the routing engine CPU.

Authentication

The `authentication-key` is an optional component. This will force the ICCP protocol to require authentication when establishing a connection. It's considered a best practice to use authentication with any type of control protocol. Authentication will prevent accidental peerings and make the environment more secure.

R3 and R4. The configuration of `R3` and `R4` is nearly identical except for the change of IP addresses. Let's review the ICCP configuration of `R3`:

```
protocols {
    iccp {
        local-ip-addr 10.8.0.6;
        peer 10.8.0.7 {
            redundancy-group-id-list 1;
            liveness-detection {
                minimum-interval 150;
                multiplier 3;
                single-hop;
            }
            authentication-key "$9$GXjkPFnCBIc5QIcylLXUjH"; ## SECRET-DATA
        }
    }
}
```

All of the ICCP components remain the same on R3 and R4 with the same `redundancy-group-id-list`, `liveness-detection`, and `authentication-key`. The only difference is that the `local-ip-addr` and `peer` have been changed to use the 10.8.0.6/31 network that sits between R3 and R4.

ICCP Verification. Now that ICCP has been configured, let's verify that it is up and operational. The `show iccp` will show more detail:

```
{master}
dhanks@R1-RE0>show iccp

Redundancy Group Information for peer 10.8.0.1
  TCP Connection       : Established
  Liveliness Detection : Up
  Redundancy Group ID          Status
    1                           Up

Client Application: l2ald_iccpd_client
  Redundancy Group IDs Joined: 1

Client Application: lacpd
  Redundancy Group IDs Joined: 1

Client Application: MCSNOOPD
  Redundancy Group IDs Joined: None
```

The TCP connection has been `Established` and ICCP is working properly. The Liveliness detection is showing `Up` as well. Another way to verify that BFD is up is via the `show bfd sessions` command:

```
{master}
dhanks@R1-RE0>show bfd session extensive
                                         Detect   Transmit
Address         State     Interface      Time     Interval  Multiplier
10.8.0.1        Up        ae0.1          0.450     0.150        3
 Client ICCP realm 10.8.0.1, TX interval 0.150, RX interval 0.150
 Client ISIS L2, TX interval 0.150, RX interval 0.150
 Session up time 6d 02:41, previous down time 00:04:17
 Local diagnostic CtlExpire, remote diagnostic CtlExpire
 Remote state Up, version 1
 Replicated
 Min async interval 0.150, min slow interval 1.000
 Adaptive async TX interval 0.150, RX interval 0.150
 Local min TX interval 0.150, minimum RX interval 0.150, multiplier 3
 Remote min TX interval 0.150, min RX interval 0.150, multiplier 3
```

```
      Local discriminator 4, remote discriminator 6
      Echo mode disabled/inactive
      Remote is control-plane independent
```

The BFD session between R1 and R2 is Up and has two clients: ICCP and ISIS. At this point, we can feel assured that ICCP is configured correctly and operational.

Multi-Chassis Aggregated Ethernet Interfaces

The real fun is configuring the MC-AE interfaces because this is where all of the design work comes in. Recall that Data Center 1 houses the PE routers R1 and R2, which need to be configured as active-active, and that Data Center 2 houses the PE routers R3 and R4, which need to be configured as active-standby. Each MC-AE configuration is a bit different because of the MC-LAG mode and different CE devices.

R1 and R2. R1 and R2 need to be able to support an active-active configuration with two CE devices: S1 and S2. This configuration is broken down into four sections:

R1:ae1

Figure 8-18 illustrates that the interface on R1 that is providing IEEE 802.3ad services to S1. The interface ae1 will need to be configured as active-active with a status-control of active. The mc-ae-id for S1 will be 1. Let's review the configuration for the interface ae1 on R1:

```
interfaces {
    ae1 {
        flexible-vlan-tagging;
        multi-chassis-protection 10.8.0.1 {
            interface ae0;
        }
        aggregated-ether-options {
            lacp {
                active;
                periodic fast;
                system-id 00:00:00:00:00:01;
                admin-key 1;
            }
            mc-ae {
                mc-ae-id 1;
                redundancy-group 1;
                chassis-id 0;
                mode active-active;
                status-control active;
            }
        }
        unit 0 {
            family bridge {
                interface-mode trunk;
                vlan-id-list [ 100 200 ];
            }
        }
    }
}
```

R2:ae1

Figure 8-18 illustrates that the interface on R2 that is providing IEEE 802.3ad services to S1. The interface ae1 will need to be configured as active-active with a status-control of standby. The mc-ae-id for S1 will be 1. Let's review the configuration for the interface ae1 on R2:

```
interfaces {
    ae1 {
        flexible-vlan-tagging;
        multi-chassis-protection 10.8.0.0 {
            interface ae0;
        }
        aggregated-ether-options {
            lacp {
                active;
                periodic fast;
                system-id 00:00:00:00:00:01;
                admin-key 1;
            }
            mc-ae {
                mc-ae-id 1;
                redundancy-group 1;
                chassis-id 1;
                mode active-active;
                status-control standby;
            }
        }
        unit 0 {
            family bridge {
                interface-mode trunk;
                vlan-id-list [ 100 200 ];
            }
        }
    }
}
```

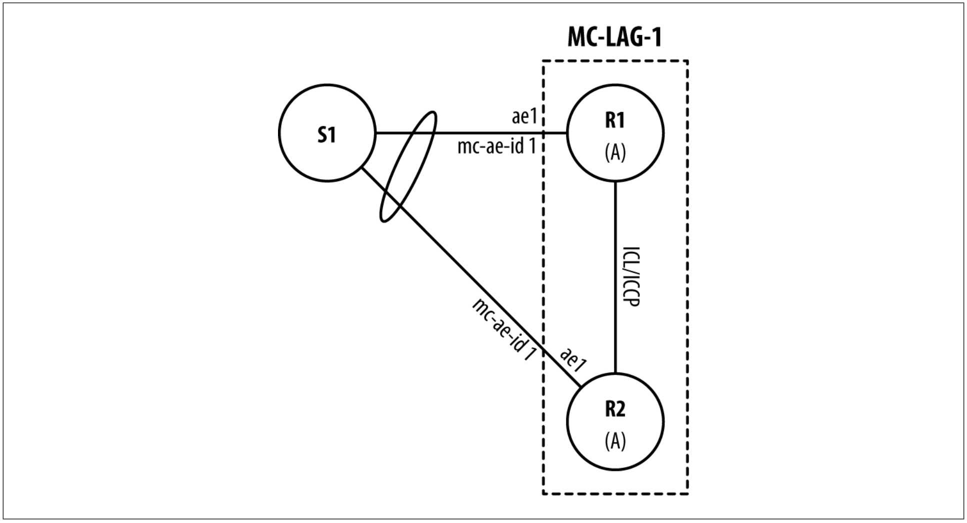


Figure 8-18. MC-LAG Case Study: R1 and R2 MC-AE-ID 1.

The only difference in the interface `ae1` MC-AE configuration between `R1` and `R2` are two components: `chassis-id` and `multi-chassis-protection`. Recall that the `chassis-id` is what separates the two PE routers in a MC-LAG configuration. One router must have a `chassis-id` of 0 while the other PE router has `chassis-id` of 1.

The `multi-chassis-protection` (MCP) must be used in an `active-active` configuration. Recall that when specifying a protected interface, it must be able to bridge the same VLAN IDs as installed on the MC-AE interface. In the case of `R1` and `R2`, each router will use its `ae0` interface which is able to bridge VLAN IDs 100 and 200:

```
interfaces {
    ae0 {
        flexible-vlan-tagging;
        aggregated-ether-options {
            minimum-links 1;
            lacp {
                active;
                periodic fast;
            }
        }
        unit 0 {
            family bridge {
                interface-mode trunk;
                vlan-id-list [ 100 200 ];
            }
        }
        unit 1 {
            vlan-id 1000;
            family inet {
                address 10.8.0.0/31;
            }
```

```
                family iso;
            }
        }
    }
```

The MCP interface will provide an Ethernet bridge between R1 and R2 in the event that frames received on R2 need to be bridged through R1 and vice versa. Don't forget that when using MC-LAG in an active-active mode, the CE device will send Ethernet frames down each link of the IEEE 802.3ad bundle; in summary, R1 and R2 will receive an equal number of frames assuming uniform distribution.

R1:ae2
: Figure 8-19 illustrates that the interface on R1 that is providing IEEE 802.3ad services to S2. The interface ae2 will need to be configured as active-active with a status-control of active. The mc-ae-id for S2 will be 2.

R2:ae2
: Figure 8-19 illustrates that the interface on R2 that is providing IEEE 802.3ad services to S2. The interface ae2 will need to be configured as active-active with a status-control of standby. The mc-ae-id for S2 will be 2.

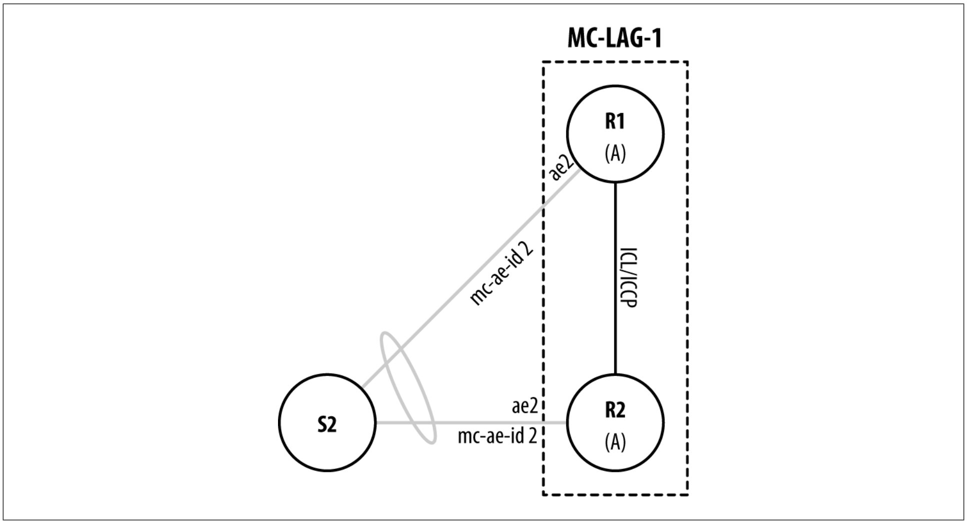


Figure 8-19. MC-LAG Case Study: R1 and R2 MC-AE-ID 2.

The MC-AE configuration for the interface ae2 and R1 and R2 is identical to configuration of interface ae1 except one component: mc-ae-id. Let's review the interface ae2 configuration of R1:

```
interfaces {
    ae2 {
        flexible-vlan-tagging;
        multi-chassis-protection 10.8.0.1 {
            interface ae0;
        }
        aggregated-ether-options {
            lacp {
                active;
                periodic fast;
                system-id 00:00:00:00:00:01;
                admin-key 1;
            }
            mc-ae {
                mc-ae-id 2;
                redundancy-group 1;
                chassis-id 0;
                mode active-active;
                status-control active;
            }
        }
        unit 0 {
            family bridge {
                interface-mode trunk;
                vlan-id-list [ 100 200 ];
            }
        }
    }
}
```

Because the interface `ae2` on `R1` and `R2` is connected to a different CE, a different `mc-ae-id` is required to create a new MC-LAG instance. Table 8-4 shows a matrix of various MC-LAG settings required for `R1` and `R2` considering there are two CE nodes, `S1` and `S2`.

Table 8-4. MC-LAG Case Study: Data Center 2 MC-AE Values.

CE	PE	Interface	MC-AE	Chassis ID	Mode	Status Control
S1	R1	ae1	1	0	active-active	active
S1	R2	ae1	1	1	active-active	standby
S2	R1	ae2	2	0	active-active	active
S2	R2	ae2	2	1	active-active	standby

Now that the MC-LAG interfaces are configured and in place, let's verify that they're up and operational. There are a couple of ways to verify. The first method it to use the `show interfaces mc-ae` command to check each of the MC-LAG interfaces:

```
{master}
dhanks@R1-RE0>show interfaces mc-ae
 Member Link                  : ae1
 Current State Machine's State: mcae active state
 Local Status                 : active
 Local State                  : up
 Peer Status                  : active
 Peer State                   : up
     Logical Interface        : ae1.0
     Topology Type            : bridge
     Local State              : up
```

```
        Peer State               : up
        Peer Ip/MCP/State        : 10.8.0.1 ae0.0 up

    Member Link                  : ae2
    Current State Machine's State: mcae active state
    Local Status                 : active
    Local State                  : up
    Peer Status                  : active
    Peer State                   : up
        Logical Interface        : ae2.0
        Topology Type            : bridge
        Local State              : up
        Peer State               : up
        Peer Ip/MCP/State        : 10.8.0.1 ae0.0 up
```

The output of the `show interfaces mc-ae` shows the status of each MC-LAG interface. As expected, the local and peer status is `active` with a state of `up`. The most important thing to verify when using an `active-active` mode is the MCP state; in this output, it's showing the peer IP address of 10.8.0.1, the correct MCP interface of `ae0.0`, and a state of `up`.

The second method is to view the status of the IEEE 802.3ad logical interfaces with `show lacp interfaces` command for the MC-LAG interface `ae1` on `R1`:

```
{master}
dhanks@R1-RE0>show lacp interfaces ae1

Aggregated interface: ae1
    LACP state:       Role   Exp   Def  Dist  Col  Syn  Aggr  Timeout  Activity
      xe-2/2/0       Actor    No    No   Yes  Yes  Yes   Yes     Fast    Active
      xe-2/2/0     Partner    No    No   Yes  Yes  Yes   Yes     Fast    Active
    LACP protocol:        Receive State  Transmit State          Mux State
      xe-2/2/0                  Current   Fast periodic Collecting distributing
```

Now let's do this again for `R2`:

```
{master}
dhanks@R2-RE0>show lacp interfaces ae1
Aggregated interface: ae1
    LACP state:       Role   Exp   Def  Dist  Col  Syn  Aggr  Timeout  Activity
      xe-2/3/0       Actor    No    No   Yes  Yes  Yes   Yes     Fast    Active
      xe-2/3/0     Partner    No    No   Yes  Yes  Yes   Yes     Fast    Active
    LACP protocol:        Receive State  Transmit State          Mux State
      xe-2/3/0                  Current   Fast periodic Collecting distributing
```

The `Mux State` of `Collecting distributing` indicates that IEEE 802.3ad has negotiated properly and that the logical interfaces are up. Extending the same method of verification to the CE, it's expected that `S1` should have a similar IEEE 802.3ad state:

```
{master:0}
dhanks@S1-RE0>show lacp interfaces
Aggregated interface: ae1
    LACP state:       Role   Exp   Def  Dist  Col  Syn  Aggr  Timeout  Activity
      xe-0/0/0       Actor    No    No   Yes  Yes  Yes   Yes     Fast    Active
      xe-0/0/0     Partner    No    No   Yes  Yes  Yes   Yes     Fast    Active
      xe-0/0/2       Actor    No    No    Yes  Yes  Yes   Yes     Fast    Active
      xe-0/0/2     Partner    No    No    Yes  Yes  Yes   Yes     Fast    Active
    LACP protocol:        Receive State  Transmit State          Mux State
      xe-0/0/0                  Current   Fast periodic Collecting distributing
      xe-0/0/2                  Current   Fast periodic Collecting distributing
```

The CE node S1 shows the logical interface ae1 in the same IEEE 802.3ad state as R1 and R2. Because of the active-active MC-LAG mode, S1 shows both child interfaces xe-0/0/0 and xe-0/0/2 as Collecting distributing. Ethernet frames will be sent to both interfaces according to the hashing characteristics of the switch. In the next example with R3 and R4, the MC-LAG will be in an active-standby state and one of the CE links will be in an Attached state; this will force Ethernet frames to egress only the active link of the IEEE 802.3ad bundle.

R3 and R4. With such an exhaustive review of the R1 and R2 configurations, there's no need to repeat the same for R3 and R4. Instead let's focus on the differences of R3 and R4:

- R3 and R4 will use a MC-LAG mode of active-standby.
- The VLAN IDs will be 300 and 400.

Table 8-5. MC-LAG Case Study: Data Center 2 MC-AE Values.

CE	PE	Interface	MC-AE	Chassis ID	Mode	Status Control
S3	R3	ae1	1	0	active-standby	active
S3	R4	ae1	1	1	active-standby	standby
S4	R3	ae2	2	0	active-standby	active
S4	R4	ae2	2	1	active-standby	standby

The good thing about using a completely different pair of PE routers is that you can recycle the MC-AE numbers. Notice that the MC-AE numbers are exactly the same from the previous configuration of R1 and R2. The only difference is the MC-LAG mode is now active-standby. Let's take a look at the interface ae1 configuration on R3:

```
interfaces {
    ae1 {
        aggregated-ether-options {
            lacp {
                active;
                periodic fast;
                system-id 00:00:00:00:00:02;
                admin-key 2;
            }
            mc-ae {
                mc-ae-id 1;
                redundancy-group 1;
                chassis-id 0;
                mode active-standby;
                status-control active;
            }
        }
```

```
            unit 0 {
                family bridge {
                    interface-mode trunk;
                    vlan-id-list [ 300 400 ];
                }
            }
        }
    }
```

There are some other operational differences such as the lacp system-id, admin-key and vlan-id-list, but the real change is putting the R3 and R4 pair into a MC-LAG mode of active-standby.

Using the same MC-LAG verification commands as before, let's inspect the state of the MC-AE interfaces on R3:

```
dhanks@R3>show interfaces mc-ae
 Member Link                  : ae1
 Current State Machine's State: mcae active state
 Local Status                 : active
 Local State                  : up
 Peer Status                  : standby
 Peer State                   : up
     Logical Interface        : ae1.0
     Topology Type            : bridge
     Local State              : up
     Peer State               : up
     Peer Ip/MCP/State        : N/A

 Member Link                  : ae2
 Current State Machine's State: mcae active state
 Local Status                 : active
 Local State                  : up
 Peer Status                  : standby
 Peer State                   : up
     Logical Interface        : ae2.0
     Topology Type            : bridge
     Local State              : up
     Peer State               : up
     Peer Ip/MCP/State        : N/A
```

Both MC-AE interfaces ae1 and ae2 are up and active from the perspective of R3. The big difference is the lack of MCP state due to the active-standby mode. Another artifact of the active-standby mode is Peer Status of standby. When verifying the MC-AE status from the point of view of the active PE router, the Peer Status should always be standby. Thus the opposite behavior should be present on R4:

```
dhanks@R4>show interfaces mc-ae
 Member Link                  : ae1
 Current State Machine's State: mcae standby state
 Local Status                 : standby
 Local State                  : up
 Peer Status                  : active
 Peer State                   : up
     Logical Interface        : ae1.0
```

```
        Topology Type            : bridge
        Local State              : up
        Peer State               : up
        Peer Ip/MCP/State        : N/A

    Member Link                  : ae2
    Current State Machine's State: mcae standby state
    Local Status                 : standby
    Local State                  : up
    Peer Status                  : active
    Peer State                   : up
        Logical Interface        : ae2.0
        Topology Type            : bridge
        Local State              : up
        Peer State               : up
        Peer Ip/MCP/State        : N/A
```

Just as suspected. `R4` shows the MC-AE interfaces `ae1` and `ae2` with a `Local Status` of `standby` with a `Peer Status` of `active`.

Given that `R3` is `active` and `R4` is `standby`, it's logical to assume that only one of the child links from the point of view of `S3` would be capable of forwarding traffic. This can be easily verified with `show lacp interfaces ae1`:

```
{master:0}
dhanks@S3>show lacp interfaces ae1
Aggregated interface: ae1
    LACP state:       Role   Exp   Def  Dist  Col  Syn  Aggr  Timeout  Activity
      xe-0/1/0       Actor    No    No   Yes  Yes  Yes   Yes     Fast    Active
      xe-0/1/0     Partner    No    No   Yes  Yes  Yes   Yes     Fast    Active
      xe-0/1/1       Actor    No    No    No   No  Yes   Yes     Fast    Active
      xe-0/1/1     Partner    No    No    No   No   No   Yes     Fast    Active
    LACP protocol:        Receive State  Transmit State          Mux State
      xe-0/1/0                  Current   Fast periodic Collecting distributing
      xe-0/1/1                  Current   Fast periodic           Attached
```

Recall that in an `active-standby` MC-LAG configuration, one of the child links in an IEEE 802.3ad bundle has a `Mux State` of `Attached`, which signals the CE not to forward traffic on that particular interface. In the example output from `S3`, it's apparent that interface xe-0/1/1 is connected to the MC-LAG standby node `R4`.

Connectivity Verification

At this point, the case study topology has been constructed and the MC-LAG components have been verified. Now would be a good time to create a baseline and verify connectivity throughout the topology. A simple but effective method to test connectivity is using the ping command sourced and destined to different CE devices. For example, to test connectivity within Data Center 1, the ping would be sourced from `S1` (192.0.2.4) and destined to `S2` (192.0.2.5). Another example is to test connectivity from Data Center 1 to Data Center 2; the ping would be sourced from `S1` (192.0.2.4) and destined to S4 (192.0.2.133). Let's take a closer look at intradata center and interdata center traffic flows.

Intradata Center Verification

The first test will source ping traffic from S1 and have a destination of S2. This will keep the traffic within the same broadcast domain and data center. There are a few things to keep in mind before starting the test:

- Data Center 1 uses an active-active MC-LAG configuration.
- S1 has two forwarding paths from the point of view of interface ae1: one link goes to R1, whereas the other goes to R2.
- R1 and R2 will use ICCP to facilitate MAC address learning.

Armed with this information, it's logical to assume that S1 will hash all egress traffic and it will be split uniformly between R1 and R2. Figure 8-20 illustrates the two possible paths for frames egressing S1: the frame could be transmitted to either R1 or R2. From the point of view of MC-LAG, each operation is a per-hop behavior; there's no local bias. For example, if R1 received an Ethernet frame from S1, R1 will bridge the frame according to its local forwarding table. Likewise, if R2 received an Ethernet frame from S1, R2 will bridge the frame according to its local forwarding table.

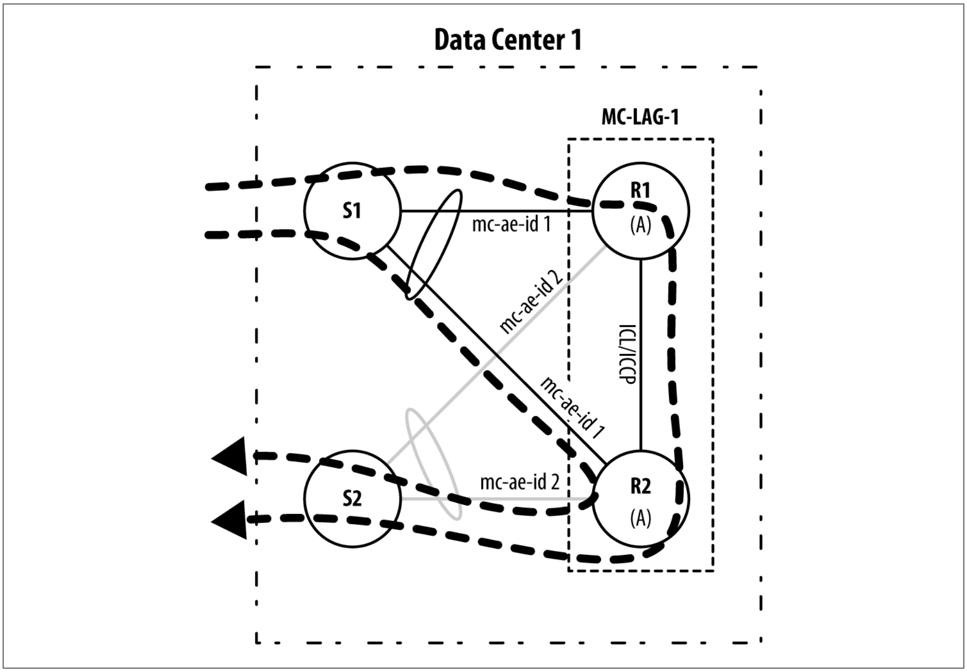


Figure 8-20. MC-LAG Case Study: Intradata Center Packet Paths.

Let's begin the test on S1 and initiate a ping destined to S2. To keep things simple, a ping count of five will be used along with the rapid option:

```
{master:0}
dhanks@S1-RE0>ping 192.0.2.5 count 5 rapid
PING 192.0.2.5 (192.0.2.5): 56 data bytes
!!!!!
--- 192.0.2.5 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max/stddev = 1.399/2.476/6.034/1.790 ms
```

Obviously, the ping was successful, but the more interesting question is how did R1 and R2 respond? Let's investigate each step of the way starting with S1 and moving all the way to S2.

When sourcing a ping from S1, it will use the system default MAC address as the source. Let's investigate and find this value:

```
{master:0}
dhanks@S1-RE0>show chassis mac-addresses
    FPC 0   MAC address information:
      Public base address     5c:5e:ab:6c:da:80
      Public count            64
```

S1 will use the source MAC address of 5c:5e:ab:6c:da:80. Let's investigate and see what the destination MAC address of S2 (192.0.2.5) is:

```
dhanks@S1-RE0>show arp
MAC Address       Address         Name   Interface          Flags
ec:9e:cd:04:d5:d2 172.19.90.53    172.19.90.53             me0.0                none
00:10:db:c6:a7:ad 172.19.91.254   172.19.91.254            me0.0                none
00:00:5e:00:01:00 192.0.2.1       192.0.2.1                vlan.100             none
00:00:5e:00:01:00 192.0.2.2       192.0.2.2                vlan.100             none
00:00:5e:00:01:00 192.0.2.3       192.0.2.3                vlan.100             none
2c:6b:f5:38:de:c0 192.0.2.5       192.0.2.5                vlan.100             none
Total entries: 6
```

S1 believes that the MAC address for 192.0.2.5 is 2c:6b:f5:38:de:c0. Let's verify the system default MAC address for S2:

```
dhanks@SW2-RE0>show chassis mac-addresses
    FPC 0   MAC address information:
      Public base address     2c:6b:f5:38:de:c0
      Public count            64
```

Cross-checking has verified the source and destination MAC addresses of S1 and S2.

- S1 MAC address is 5c:5e:ab:6c:da:80.
- S2 MAC address is 2c:6b:f5:38:de:c0.

Armed with this data, let's see how the per-hop behavior is working from the perspective of R1. The best method is to look at the MAC address table and see which MAC address were learned locally and remotely:

```
1    {master}
2    dhanks@R1-RE0>show bridge mac-table
3
4    MAC flags (S -static MAC, D -dynamic MAC, L -locally learned
5               SE -Statistics enabled, NM -Non configured MAC, R -Remote PE MAC)
6
7    Routing instance : default-switch
8     Bridging domain : VLAN100, VLAN : 100
9       MAC                 MAC      Logical
10        address             flags    interface
11        2c:6b:f5:38:de:c0   DR       ae2.0
12        2c:6b:f5:38:de:c2   DR       ae2.0
13        5c:5e:ab:6c:da:80   DL       ae1.0
14
15    MAC flags (S -static MAC, D -dynamic MAC, L -locally learned
16               SE -Statistics enabled, NM -Non configured MAC, R -Remote PE MAC)
```

From the perspective of R1, line 11 indicates that the destination MAC address of 2c:6b:f5:38:de:c0 was learned remotely. Line 13 indicates that the source MAC address of 5c:5e:ab:6c:da:80 was learned locally.

Let's view the same MAC table, but from the perspective of R2:

```
1    {master}
2    dhanks@R2-RE0>show bridge mac-table
3
4    MAC flags (S -static MAC, D -dynamic MAC, L -locally learned
5               SE -Statistics enabled, NM -Non configured MAC, R -Remote PE MAC)
6
7    Routing instance : default-switch
8     Bridging domain : VLAN100, VLAN : 100
9       MAC                 MAC      Logical
10        address             flags    interface
11        2c:6b:f5:38:de:c0   DL       ae2.0
12        2c:6b:f5:38:de:c2   DL       ae2.0
13        5c:5e:ab:6c:da:80   DR       ae1.0
14
15    MAC flags (S -static MAC, D -dynamic MAC, L -locally learned
16               SE -Statistics enabled, NM -Non configured MAC, R -Remote PE MAC)
```

From the perspective of R2, line 11 indicates that the destination MAC address of 2c:6b:f5:38:de:c0 was learned locally. This makes sense as S2 is directly attached to R2. Line 13 indicates that the source MAC address of 5c:5e:ab:6c:da:80 was learned remotely. In summary, the MAC learning of R1 and R2 are perfectly asymmetric.

To further investigate and confirm the remote MAC learning, let's take a look at what ICCP is reporting for MAC learning. Let's start with R1:

```
{master}
dhanks@R1-RE0>show l2-learning redundancy-groups remote-macs

Redundancy Group ID : 1     Flags : Local Connect,Remote Connect

Service-id  Peer-Addr    VLAN        MAC            MCAE-ID  Subunit  Opcode   Flags
Status
```

```
1          10.8.0.1    100    2c:6b:f5:38:de:c0  2       0       1       0      Installed
1          10.8.0.1     100     2c:6b:f5:38:de:c2   2       0        1        0
Installed
```

This confirms that the MAC address 2c:6b:f5:38:de:c0 was learned via ICCP from `R2` and showing as `Installed` on `R1`.

Now let's confirm the remote MAC learning from the perspective of R2:

```
{master}
dhanks@R2-RE0>show l2-learning redundancy-groups remote-macs

Redundancy Group ID : 1     Flags : Local Connect,Remote Connect

Service-id  Peer-Addr    VLAN       MAC            MCAE-ID  Subunit  Opcode    Flags
Status
1           10.8.0.0     100     5c:5e:ab:6c:da:80   1        0        1        0
Installed
```

Just as suspected. The MAC address 5c:5e:ab:6c:da:80 was learned via ICCP from `R1` and shows as `Installed` on `R2`.

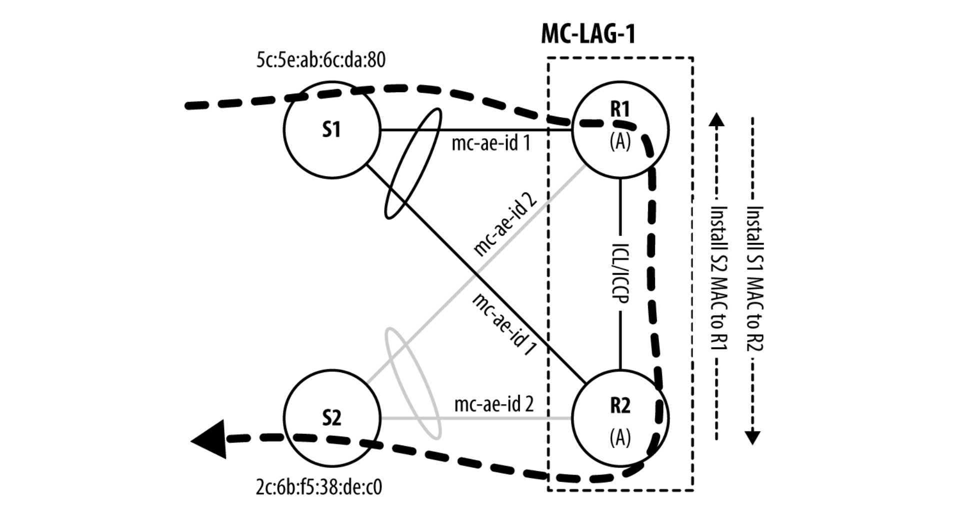


Figure 8-21. MC-LAG Case Study: Intradata Center Packet Flow Test Results.

As shown in Figure 8-21, this evidence indicates the following chain of events:

- `R1` installed the MAC address 5c:5e:ab:6c:da:80 via ICCP to `R2`.
- `R2` installed the MAC address 2c:6b:f5:38:de:c0 via ICCP to `R1`.
- `S1` hashed the ICMP traffic destined to S2 via `R1`.
- `R1` received an Ethernet frame destined to 2c:6b:f5:38:de:c0. This MAC address exists in the forwarding table because it was learned via ICCP from `R2`.

- R1 bridges the Ethernet frame destined to 2c:6b:f5:38:de:c0 to R2.
- R2 receives the Ethernet frame destined to 2c:6b:f5:38:de:c0. This MAC address exists in the forwarding table because it was learned locally from S2.
- R2 bridges the Ethernet frame destined to 2c:6b:f5:38:de:c0 to S2.
- S2 receives the Ethernet frame.

Although it's possible for S1 to hash Ethernet frames to R2, this example illustrates that the hashing function of S1 decided to only use R1 during the ping test. With a more diverse set of flows, the hashing function on S1 would have more data to work with and would thus ultimately have uniform traffic distribution between R1 and R2.

Interdata Center Verification

The final piece of verification is testing the interdata center connectivity. This scenario will require the packet to be sourced from a CE in Data Center 1 and be destined to a CE in Data Center 2. This creates an interesting test case as the packet can take different paths at different sections in the topology based on hashing algorithms, as illustrated in Figure 8-22.

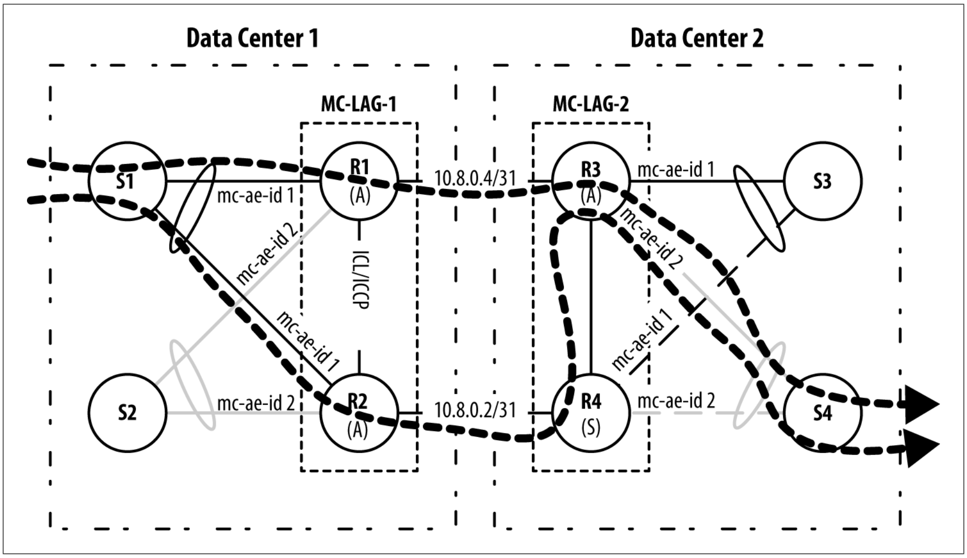


Figure 8-22. MC-LAG Case Study: Interdata Center Possible Packet Flows.

In this specific test case, traffic will be sourced from S1 (192.0.2.4) and destined to S4 (192.0.2.133). Having the source and destination on opposite ends of the topology and diagonally opposed creates an interesting packet flow. Recall that MC-LAG on R3 and R4 are configured for `active-standby`. Because in this example the destination is S4, it creates an interesting bifurcation between R3 and R4. If R3 receives an Ethernet frame destined to S4, it's able to forward the frame directly to S4. However, if R4 receives an

Ethernet frame destined to S4—because of the nature of active-standby MC-LAG—R4 will have to forward the frame to R3, because R4's link to S4 is in standby. Once R4 forwards the frame to R3, it can then be forwarded to its final destination of S4.

Let's begin the test case and execute the ping from S1. Just like the previous intradata center test, the rapid option will be used with a count of five:

```
master:0}
dhanks@S1-RE0>ping source 192.0.2.4 192.0.2.133 count 5 rapid
PING 192.0.2.133 (192.0.2.133): 56 data bytes
!!!!!
--- 192.0.2.133 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max/stddev = 1.175/2.255/3.436/0.956 ms
```

No surprise that the ping worked. Let's begin to break it down. Because the destination address of 192.0.2.133 is on a different subnet than the source address of 192.0.2.4, S1 will have to route the traffic. Let's take a look at the RIB:

```
{master:0}
dhanks@S1-RE0>show route 192.0.2.133

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

192.0.2.128/26     *[Static/5] 00:53:52
                    > to 192.0.2.1 via vlan.100
```

There's a static route 192.0.2.128/26 that points to the next-hop of 192.0.2.1. Let's find the MAC address and egress interface of 192.0.2.1:

```
1    {master:0}
2    dhanks@S1-RE0>show arp
3    MAC Address       Address         Name                    Interface           Flags
4    ec:9e:cd:04:d5:d2 172.19.90.53     172.19.90.53            me0.0               none
5    00:10:db:c6:a7:ad 172.19.91.254    172.19.91.254           me0.0               none
6    00:00:5e:00:01:00 192.0.2.1        192.0.2.1               vlan.100            none
7    00:00:5e:00:01:00 192.0.2.2        192.0.2.2               vlan.100            none
8    00:00:5e:00:01:00 192.0.2.3        192.0.2.3               vlan.100            none
9    2c:6b:f5:38:de:c0 192.0.2.5        192.0.2.5               vlan.100            none
10    Total entries: 6
```

As shown on line 6, the MAC address for 192.0.2.1 is 00:00:5e:00:01:00. Now let's find which interface will be used for egress:

```
1    {master:0}
2    dhanks@S1-RE0>show ethernet-switching table
3    Ethernet-switching table: 6 entries, 4 learned
4      VLAN             MAC address       Type         Age Interfaces
5      vlan_100         *                 Flood          - All-members
6      vlan_100         00:00:5e:00:01:00 Learn          0 ae1.0
7      vlan_100         00:1f:12:b8:8f:f0 Learn          0 ae1.0
8      vlan_100         2c:6b:f5:38:de:c0 Learn       3:27 ae1.0
9      vlan_100         2c:6b:f5:38:de:c2 Learn          0 ae1.0
10      vlan_100         5c:5e:ab:6c:da:80 Static         - Router
```

Again, line 6 shows that the egress interface for Ethernet frames destined to 00:00:5e:00:01:00 will egress interface `ae1.0`. Recall that interface `ae1` on `S1` has two member links; one link is connected to `R1`, while the other link is connected to `R2`.

```
{master:0}
dhanks@S1-RE0>show lacp interfaces
Aggregated interface: ae1
    LACP state:       Role    Exp   Def  Dist  Col  Syn  Aggr  Timeout  Activity
      xe-0/0/0       Actor     No    No   Yes  Yes  Yes   Yes     Fast    Active
      xe-0/0/0     Partner     No    No   Yes  Yes  Yes   Yes     Fast    Active
      xe-0/0/2       Actor     No    No   Yes  Yes  Yes   Yes     Fast    Active
      xe-0/0/2     Partner     No    No   Yes  Yes  Yes   Yes     Fast    Active
    LACP protocol:        Receive State  Transmit State          Mux State
      xe-0/0/0                  Current   Fast periodic Collecting distributing
      xe-0/0/2                  Current   Fast periodic Collecting distributing
```

Without being able to predict if the ping traffic was destined to `R1` or `R2`, let's use the `traceroute` command to see which routers are between `S1` and `S4`:

```
{master:0}
dhanks@S1-RE0>traceroute source 192.0.2.4 192.0.2.133
traceroute to 192.0.2.133 (192.0.2.133) from 192.0.2.4, 30 hops max, 40 byte packets
 1  192.0.2.3 (192.0.2.3)  3.841 ms  4.931 ms  0.678 ms
 2  10.8.0.3 (10.8.0.3)  1.417 ms  7.274 ms  0.737 ms
 3  192.0.2.133 (192.0.2.133)  1.705 ms  1.486 ms  1.493 ms
```

Using the `traceroute` test, it's evident that the path used is `S1` → `R2` → `R4` → `S4`. However, looks can be deceiving. Let's continue investigating the path of traffic flow from `S1` to `S4`.

Assuming that traffic from `S1` was sent to `R2`, let's view the RIB of `R2` to find the next-hop:

```
{master}
dhanks@R2-RE0>show route 192.0.2.133

inet.0: 22 destinations, 22 routes (22 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

192.0.2.128/26     *[IS-IS/18] 00:56:34, metric 73
                    > to 10.8.0.3 via ae3.0
```

Recall that the 192.0.2.128/26 network is advertised by both `R3` and `R4` and that the shortest path to 192.0.2.128/26 is via 10.8.0.3 to `R4`. Let's move to `R4` and look at the MAC address table, but first the MAC address of 192.0.2.133 must be found:

```
{master:0}
dhanks@S4>show chassis mac-addresses
    FPC 0   MAC address information:
      Public base address     00:19:e2:57:b6:40
      Public count            128
```

Recall that the `traceroute` indicated `R4` was the last router before arriving at the final destination of 192.0.2.133. Now that the MAC address of `S4` has been identified, let's take a look at the MAC table of `R4`:

```
dhanks@R4>show bridge mac-table 00:19:e2:57:b6:40
```

```
MAC flags (S -static MAC, D -dynamic MAC, L -locally learned
           SE -Statistics enabled, NM -Non configured MAC, R -Remote PE MAC)

Routing instance : default-switch
 Bridging domain : VLAN300, VLAN : 300
   MAC                 MAC      Logical
   address             flags    interface
   00:19:e2:57:b6:40   D        ae0.0
```

Now that's interesting. According to the previous `traceroute`, the path from `S1` to `S4` was `S1` → `R2` → `R4` → `S4`. But clearly through investigation and verification, the path was actually `S1` → `R2` → `R4` → `R3` → `S4`. Recall that `R4`'s link to `S4` is in standby mode and cannot be used for forwarding:

```
dhanks@R4>show lacp interfaces ae2
Aggregated interface: ae2
    LACP state:       Role   Exp   Def  Dist  Col  Syn  Aggr  Timeout  Activity
      xe-2/0/1       Actor    No    No    No   No   No   Yes     Fast    Active
      xe-2/0/1     Partner    No    No    No   No  Yes   Yes     Fast    Active
    LACP protocol:        Receive State  Transmit State          Mux State
      xe-2/0/1                  Current   Fast periodic            Waiting
```

The interface `xe-2/0/1` on `R4` is in a `Mux State` of `Waiting`; this state indicates that the interface `xe-2/0/1` isn't available to forward traffic. Because the interface `xe-2/0/1` isn't available to forward traffic, the only other option is to use the link between `R3` and `R4`. In this example, `R4` must use the interface `ae0.0` to forward Ethernet frames destined to 00:19:e2:57:b6:40. The interface `ae0.0` on `R4` is directly connected to `R3`. Let's take a look at the MAC table on `R3` to find the forwarding path for 00:19:e2:57:b6:40:

```
dhanks@R3>show bridge mac-table 00:19:e2:57:b6:40

MAC flags (S -static MAC, D -dynamic MAC, L -locally learned
           SE -Statistics enabled, NM -Non configured MAC, R -Remote PE MAC)

Routing instance : default-switch
 Bridging domain : VLAN300, VLAN : 300
   MAC                 MAC      Logical
   address             flags    interface
   00:19:e2:57:b6:40   D        ae2.0
```

`R3` has an entry for 00:19:e2:57:b6:40 that points to the `ae2.0` interface; this interface is directly connected to `S4` and is able to successfully deliver the Ethernet frame. This concludes the verification and investigation of sourcing a ping from S1 to S4. At first glance, one would assume that the path would be `S1` → `R2` → `R4` → `S4`, but with careful inspection it was determined that the actual traffic path was `S1` → `R2` → `R4` → `R3` → `S4`. This is due to `R4` being in standby mode and having to use `R3` to bridge Ethernet frames destined to the CE.

Case Study Summary

The MC-LAG case study is an interesting journey. It starts out with a basic topology, but quickly builds multiple facets of MC-LAG. `R1` and `R2` were configured as `active-active`, whereas `R3` and `R4` were configured as `active-standby`. This creates an interesting topology as packets move through the network. As demonstrated in the interdata center verification, things aren't always what they seem.

As you think about implementing MC-LAG in your network, consider the details of this case study. What problems will MC-LAG solve? What MC-LAG is more suited to your network?

Summary

MC-LAG is a simple but yet effective feature that can be used to increase the performance and resiliency in your network. You're able to choose between an `active-active` or `active-standby` topology that best fits your use case. For example, when creating full mesh topology of IEEE 802.1Q links between different tiers of a network, it would make sense to use an `active-active` topology to fully utilize each of the links and also provide resiliency during a failure scenario.

ICCP is the heart and soul of MC-LAG; it leverages existing and well-understood technologies such as TCP/IP and BFD, which make the configuration and troubleshooting very easy for new users. ICCP serves as a very simple and extendable protocol that keeps track of state changes, configuration changes, and signal information between routers.

One of the most common use cases for MC-LAG is to dual-home CE devices. This comes in the form of top-of-rack switches or providing node redundancy for customer routers in a WAN environment. The benefits of MC-LAG are:

- No spanning tree is required; MC-LAG has built-in features in the PFE to detect and prevent Layer 2 loops.
- The CE implementation is transparent and only requires IEEE 802.3ad.
- The PE implementation of MC-LAG doesn't require a reboot and is less disruptive to the network.
- The design of MC-LAG allows each PE to operate independently from the other. In the event of a misconfiguration of a PE, the error would be isolated to that particular PE router and not impact the other PE. This inherently provides an additional level of high availability.

The design of MC-LAG inherently forces a per-hop behavior in a network. The advantage is that you are able to view and troubleshoot problems at every hop in a packet's journey. For example, if you are trying to determine which interface a MC-LAG router will use to forward an Ethernet frame, you can use standard tools such as viewing the forwarding table of the router.

Chapter Review Questions

1. What's the maximum number of PE routers that can participate in MC-LAG?
   a. 1
   b. 2
   c. 3
   d. 4
2. Can `active-active` MC-LAG be used with DPC line cards?
   a. Yes
   b. No
3. Does the `service-id` have to match between PE routers?
   a. Yes
   b. No
4. Does the `chassis-id` have to match between PE routers?
   a. Yes
   b. No
5. What's the purpose of `mc-ae-id`?
   a. A unique identifier to partition different MC-LAG interfaces
   b. A unique identifier to group together interfaces across different PE routers to form a single logical interface facing towards the CE
   c. A unique identifier for each physical interface
   d. A unique identifier for each routing instance
6. What is a `redundancy-group`?
   a. A collection of MC-LAG interfaces
   b. Serves as a broadcast medium for applications between PE routers
   c. Provides physical interface redundancy
   d. Used by ICCP to update MAC addresses
7. Which MC-LAG mode requires an ICL?
   a. `active-active`
   b. `active-standby`
8. How does MC-LAG `active-standby` influence how the CE forwards traffic?
   a. Administratively disables one of the interfaces
   b. Physically shuts down one of the interfaces
   c. Removes one of the interfaces from the LACP bundle
   d. Places one of the interfaces of the CE into a Mux State of "Attached"

9. Which feature provides more routing engine scale?
   a. MC-LAG
   b. MX-VC
10. What IFFs are supported on MC-LAG interfaces (as of Junos 11.4)?
    a. inet
    b. bridge
    c. vpls
    d. ccc

Chapter Review Answers

1. **Answer: B**. As of Junos 11.4, only two PE routers can participate in MC-LAG.
2. **Answer: B**. When using MC-LAG in `active-active` mode, you must use the Trio-based MPC line cards.
3. **Answer: A**. Recall that the `service-id` is used to uniquely identify routing instances across PE routers when forming a MC-LAG instance. Although as of Junos 11.4, MC-LAG d doesn't support routing instances; the `service-id` must match between both PE routers under `[switch-options service-id]`.
4. **Answer: B**. The `chassis-id` is used to uniquely identify each PE router when forming a MC-LAG instance. Each PE router needs a different `chassis-id`.
5. **Answer: A,B**. The `mc-ae-id` has two purposes: glue together MC-LAG interfaces across different PE routers and uniquely identify different logical MC-LAG interfaces. For example, if `CE1` was connected to `PE1:xe-0/0/0` and `PE2:xe-0/0/0`, the `mc-ae-id` would be 1. If `CE2` was connected to `PE1:xe-0/0/1` and `PE2:xe-0/0/1`, the `mc-ae-id` would be 2.
6. **Answer: A,B,D**. Redundancy groups are a collection of MC-AE IDs that share the same VLAN IDs. Redundancy groups act as a broadcast medium between PE routers so that application messages are concise and efficient. In this example, the application would be MAC address updates.
7. **Answer: A**. Only MC-LAG in `active-active` mode requires an ICL. The ICL is used as a Layer 2 link between the two PE routers so that as frames arrive from the CE, both the PE routers can handle the frames efficiently.
8. **Answer: D**. The MC-LAG node that's currently in the state of `standby` will signal to the CE to place its member link in the `Mux State` of `Attached`. This will prevent the CE from forwarding frames to the standby link.
9. **Answer: A**. Because MC-LAG requires a control plane per chassis, it will inherently offer more scale per chassis. On the other hand, MX-VC creates a single logical control plane and the scale per chassis is reduced.

10. **Answer: B,C**. As of Junos 11.4, only the bridge and VPLS interface families are supported. CCC is supported, but only as an encapsulation type. Recall that `family ccc` is for LSP stitching.

CHAPTER 9

Junos High Availability on MX Routers

This chapter covers Junos software high-availability (HA) features supported on MX routers. These HA features serve to lessen, if not negate, the impacts of software or routing engine (RE) faults that would otherwise disrupt network operators and potential impact revenue.

The topics discussed in this chapter include:

- Junos HA feature Overview
- Graceful Routing Engine Switchover
- Graceful Restart
- Nonstop Routing and Nonstop Bridging
- In-Service Software Upgrades
- ISSU demonstration

Junos High-Availability Feature Overview

Numerous HA features have been added to Junos software in the 13 years since its first commercial release. These enhancements were in keeping with the ever more critical role that networks play in the infrastructure of modern society. As competition increased in the networking industry, both Enterprise and Service Providers demanded HA features so that they, in turn, could offer premium services that included network reliability as a critical component of their Service Level Agreements (SLAs).

Gone are the days of hit-and-miss network availability, whether caused by failing hardware or faulty software; people just can’t get their jobs done without computers, and for most people, a computer these days is only as useful as its network connection. This is the era of “five nines” reliability and achieving that goal on a yearly basis can be demanding. When luck is simply not enough, it’s good to know that MX routers have inherited the complete suite of field-proven Junos HA features, which, when combined with sound network design principals, enable HA.

This chapter starts with a brief overview of the HA features available on all MX routers; subsequent sections will detail the operation, configuration, and usage of these features.

Graceful Routing Engine Switchover

Graceful Routing Engine Switchover (GRES) is the foundation upon which most other Junos HA features are stacked. The feature is only supported on platforms that have support for dual REs. Here, the term graceful refers to the ability to support a change in RE mastership without forcing a reboot of the PFE components. Without GRES, the new master RE reboots the various PFE components to ensure it has consistent PFE state. A PFE reboot forces disruptions to the data plane, a hit that can last several minutes while the component reboots and is then repopulated with current routing state.

Graceful Restart

Graceful Restart (GR) is a term used to describe protocol enhancements designed to allow continued dataplane forwarding in the face of a routing fault. GR requires a stable network topology (for reasons described in the following), requires protocol modifications, and expects neighboring routers to be aware of the restart and to assist the restarting router back into full operation. As a result, GR is not transparent and is losing favor to Nonstop Routing. Despite these shortcomings, it's common to see GR on platforms with a single RE as GR is the only HA feature that does not rely on GRES.

Nonstop Routing

Nonstop Routing (NSR) is the preferred method for providing hitless RE switchover. NSR is a completely internal solution, which means it requires no protocol extensions or interactions from neighboring nodes. When all goes to the NSR plan, RE mastership switches with no dataplane hit or externally visible protocol reset; to the rest of the network, everything just keeps working as before the failover.

Because GR requires external assistance and protocol modifications, whereas NSR does not, the two solutions are somewhat diametrically opposed. This means you must choose either GR or NSR as you cannot configure full implementations of both simultaneously. It's no surprise when one considers that GR announces the control plane reset and asks its peers for help in ramping back up, while NSR seeks to hide such events from the rest of the world, that it simply makes no sense to try and do both at the same time!

Nonstop Bridging

Nonstop Bridging (NSB) adds hitless failover to Layer 2 functions such as MAC learning and to Layer 2 control protocols like spanning tree and LLDP. Currently, NSB is available on MX and supported EX platforms.

In-Service Software Upgrades

In-Service Software Upgrades (ISSU) is the capstone of Junos HA. The feature is based on NSR and GRES, and is designed to allow the user to perform software upgrades that are virtually hitless to the dataplane while being completely transparent in the control plane. Unlike a NSR, a small dataplane hit (less than five seconds) is expected during an ISSU as new software is loaded into the PFE components during the process.

Graceful Routing Engine Switchover

As noted previously, GRES is a feature that permits Juniper routers with dual REs to perform a switch in RE mastership without forcing a PFE reset. This permits uninterrupted dataplane forwarding, but unless combined with GR or NSR, does not in itself preserve control plane or forwarding state.

The foundation of GRES is kernel synchronization between the master and backup routing engines using Inter-Process Calls (IPC). Any updates to kernel state that occur on the master RE, for example to reflect a changed interface state or the installation of a new next-hop, are replicated to the backup RE as soon as they occur and before pushing the updates down into other parts of the system, for example, to the FPCs. If the kernel on the master RE stops operating, experiences a hardware failure, a configured process is determined to be thrashing, or the administrator initiates a manual switchover, mastership switches to the backup RE.

Performing a switchover before the system has synchronized leads to an all-bets-off situation. Those PFE components that are synchronized are not reset, while the rest of the components are. Junos enforces a GRES holddown timer that prevents rapid back-to-back switchovers, which seems to be all the rage in laboratory testing. The 240-second (4-minute) timer between manually triggered GRES events is usually long enough to allow for complete synchronization, and therefore helps to ensure a successful GRES event. The holddown timer is not enforced for automatic GRES triggers such as a hardware failure on the current master RE. If you see the following, it means you need to cool your GRES jets for a bit to allow things to stabilize after an initial reboot or after a recent mastership change:

```
{backup}
jnpr@R1-RE0>request chassis routing-engine master acquire no-confirm
Command aborted. Not ready for mastership switch, try after 234 secs.
```

The GRES Process

The GRES feature has three main components: synchronization, switchover, and recovery.

Synchronization

GRES begins with synchronization between the master and backup RE. By default after a reboot, the RE in slot 0 becomes the master. You can alter this behavior, or disable a given RE if you feel it’s suffering from a hardware malfunction or software corruption, at the `[edit chassis redundancy]` hierarchy:

```
{master}[edit]
jnpr@R1-RE1# set chassis redundancy routing-engine 1 ?
Possible completions:
  backup               Backup Routing Enguine
  disabled             Routing Engine disabled
  master               Master Routing Engine
{master}[edit]
jnpr@R1-RE1# set chassis redundancy routing-engine 1
```

When the BU RE boots, it starts the kernel synchronization daemon called ksyncd. The ksyncd process registers as a peer with the master kernel and uses IPC messages to carry routing table socket (rtsock) messages that represent current master kernel state. Synchronization is considered complete when the BU RE has matching kernel state. Once synchronized, ongoing state changes in the master kernel are first propagated to the backup kernel before being sent to other system components. This process helps ensure tight coupling between the master and BU kernels as consistent kernel state is critical to the success of a GRES. Figure 9-1 shows the kernel synchronization process between a master and BU RE.

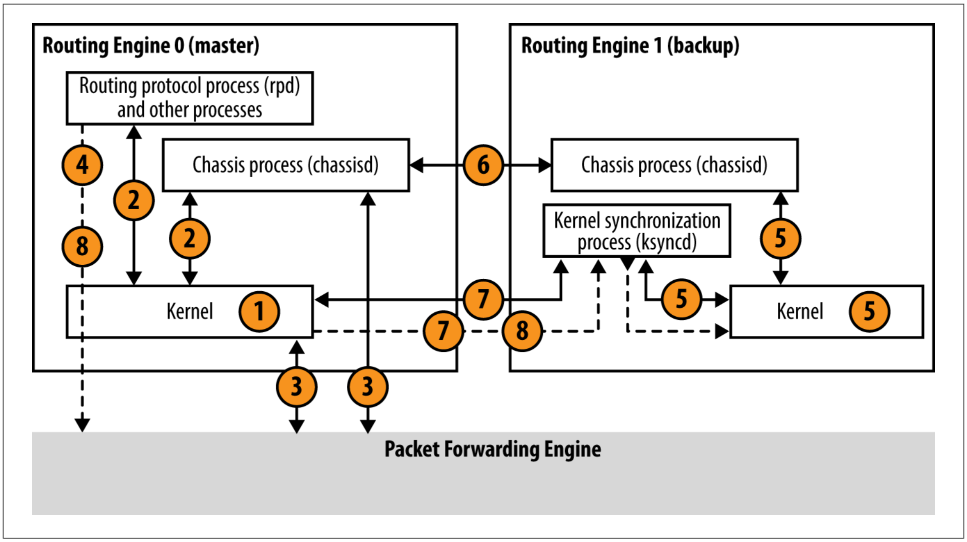

Routing Engine 0 (master)
Routing protocol process (rpd) and other processes
Chassis process (chassisd)
Kernel
1
2
2
4
8
3
3
6
7
7
8
Routing Engine 1 (backup)
Chassis process (chassisd)
Kernel synchronization process (ksyncd)
Kernel
5
5
5
Packet Forwarding Engine

Figure 9-1. GRES and Kernel Synchronization.

The steps in Figure 9-1 show the major GRES processing steps after a reboot.

1. The master RE starts. As noted previously, by default this is RE0 but can be altered via configuration.
2. The various routing platform processes, such as the chassis process (chassisd), start.
3. The Packet Forwarding Engine starts and connects to the master RE.
4. All state information is updated in the system.

5. The backup RE starts.
6. The system determines whether graceful RE switchover has been enabled.
7. The kernel synchronization process (`ksyncd`) synchronizes the backup RE with the master RE.
8. After `ksyncd` completes the synchronization, all state information and the forwarding table are updated.

Routing Engine Switchover

RE switchover can occur for a variety of reasons. These include the following:

- By having the `chassid` process monitor for loss of keepalive messages from master RE for 2 seconds (4 seconds on the now long-in-the-tooth M20 routers). The keepalive process functions to ensure that a kernel crash or RE hardware failure on the current master is rapidly detected without need for manual intervention.
- Rebooting the current master.
- By having the `chassid` process on the BU RE monitor chassis FPGA mastership state and becoming master whenever the chassis FPGA indicates there is no current master.
- By detecting a failed hard disk or a thrashing software process, when so configured, as neither is a default switchover trigger.
- When instructed to perform a mastership change by the operating issuing a `request chassis routing-engine switchover` command. This command is the preferred way to force a mastership change during planned maintenance windows, and for GRES feature testing in general.

See the section on NSR for details on other methods that can be used to induce a GRES event when testing HA features.

Upon seizing mastership, the new master's `chassisd` does not restart FPCs. During the switchover, protocol peers may detect a lack of protocol hello/keepalive messages, but this window is normally too short to force a protocol reset; for example, BGP needs to miss three keepalives before its hold-time expires. In addition, the trend in Junos is to move protocol-based peer messages, such as OSPF's hello packets, into the PFE via the `ppmd` daemon, where they are generated independently of the RE. PFE-based message generation not only improves scaling and accommodates lower hello times, but also ensures that protocol hello messages continue to be sent through a successful GRES event. However, despite the lack of PFE reset, protocol sessions may still be reset depending on whether GR or NSR is also configured in addition to basic GRES. Stating this again, GRES alone cannot prevent session reset, but it does provide the infrastructure needed to allow GR and NSR control plane protection, as described in later sections.

After the switchover, the new master uses the BSD init process to start/restart daemons that wish to run only when the RE is a master and the PFEs reestablish their connections with `chassisd`. The `chassisd` process then relearns and validates PFE state as needed by querying its peers in the PFE.

Figure 9-2 shows the result of the switchover process.

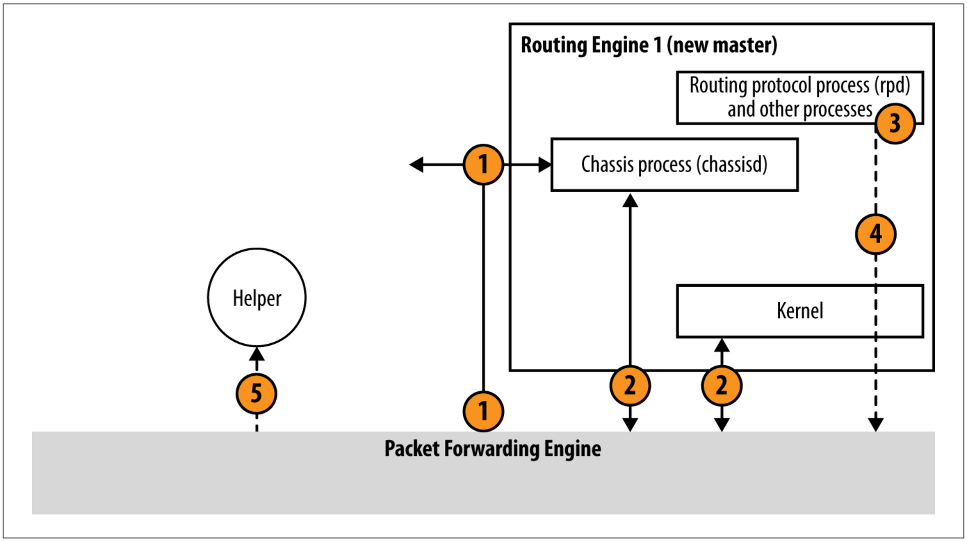

Routing Engine 1 (new master)
Routing protocol process (rpd) and other processes
3
1
Chassis process (chassisd)
4
Helper
Kernel
5
1
2
2
Packet Forwarding Engine

Figure 9-2. The Routing Engine Switchover Process.

The numbers in Figure 9-2 call out the primary sequence of events that occur as part of a GRES-based RE switchover:

1. Loss of keepalives (or other stimulus) causes `chassid` to gracefully switch control to the current backup RE.
2. The Packet Forwarding Engine components reconnect to the new master RE.
3. Routing platform processes that are not part of graceful RE switchover, and which only run on a master RE, such as the routing protocol process (`rpd`) when NSR is not in effect, restart.
4. Any in-flight kernel state information from the point of the switchover is replayed, and the system state is once again made consistent. Packet forwarding and FIB state is not altered, and traffic continues to flow as it was on the old master before the switchover occurred.

5. When enabled, Graceful Restart (GR) protocol extensions collect and restore routing information from neighboring peer helper routers. The role of helper routers in the GR process is covered in a later section.

What Can I Expect after a GRES?

Table 9-1 details the expected outcome for a RE switchover as a function of what mix of HA features are, or are not, configured at the time.

Table 9-1. Expected GRES Results.

Feature	Effect	Notes
Redundant RE, no HA features	PFE reboot and the control plane reconverges on new master	All physical interfaces are taken offline, Packet Forwarding Engines restart, the standby routing engine restarts the routing protocol process (`rpd`), and all hardware and interfaces are discovered by the new master RE. The switchover takes several minutes and all of the router's adjacencies are aware of the physical (interface alarms) and routing (topology) change.
GRES only	During the switchover, interface and kernel information is preserved. The switchover is faster because the Packet Forwarding Engines are not restarted.	The new master RE restarts the routing protocol process (rpd). All hardware and interfaces are acquired by a process that is similar to a warm restart. All adjacencies are aware of the router's change in state due to control plane reset.
GRES plus Graceful Restart	Traffic is not interrupted during the switchover. Interface and kernel information is preserved. Graceful restart protocol extensions quickly collect and restore routing information from the neighboring routers.	Neighbors are required to support graceful restart, and a wait interval is required. The routing protocol process (rpd) restarts. For certain protocols, a significant change in the network can cause graceful restart to stop.
GRES plus NSR/NSB	Traffic is not interrupted during the switchover. Interface, kernel, and routing protocol information is preserved for NSR/NSB supported protocols and options.	Unsupported protocols must be refreshed using the normal recovery mechanisms inherent in each protocol.

The table shows that NSR, with its zero packet loss and lack of any external control plane flap (for supported protocols), represents the best case. In the worst case, when no HA features are enabled, you can expect a full MPC (PFE) reset, and several minutes of outage (typically ranging from 4 to 15 minutes as a function of system scale), as the control plane converges and forwarding state is again pushed down into the PFE after a RE mastership change. The projected outcomes assume that the system, and the related protocols, have all converged and completed any synchronization, as needed for NSR, before a switchover occurs.

The outcome of a switchover that occurs while synchronization is still underway is unpredictable, but will generally result in dataplane and possible control plane resets, making it critical that the operator know when it's safe to perform a switchover. Knowing when its "safe to switch" is a topic that's explored in detail later in this chapter.

Though not strictly required, ruining the same Junos version on both REs is a good way to improve the odds of a successful GRES.

Configure GRES

GRES is very straightforward to configure and requires only a `set chassis redundancy graceful-switchover` statement to place it into effect. Though not required, it's recommended that you use `commit synchronize` whenever GRES is in effect to ensure consistency between the master and backup REs to avoid inconsistent operation after a RE switchover.

When you enable GRES, the system automatically sets the `chassis redundancy keepalive-time` to 2 seconds, which is the lowest supported interval; attempting to modify the keepalive value when GRES is in effect results in a commit fail, as shown.

```
jnpr@R1-RE1# show chassis
redundancy {
    ##
    ## Warning: Graceful switchover configured, cannot change the default keepalive
interval
    ##
    keepalive-time 25;
    graceful-switchover;
}
```

When GRES is disabled, you can set the keepalive timer to the range of 2 to 10,000 seconds, with 300 seconds being the non-GRES default. When GRES is disabled, you can also specify whether a failover should occur when the keepalive interval times out with the `set chassis redundancy failover on-loss-of-keepalives` statement.

However, simply enabling GRES results in two-second-fast keepalive along with automatic failover. With the minimal GRES configuration shown, you can expect automatic failover when a hardware or kernel fault occurs on the master RE resulting in a lack of keepalives for two seconds:

```
 [edit]
jnpr@R1-RE1# show chassis
redundancy {
    graceful-switchover;
}
```

Note how the banner changes to reflect master or backup status once GRES is committed:

```
[edit]
jnpr@R1-RE1# commit
commit complete

{master}[edit]
jnpr@R1-RE1#
```

And, once in effect, expect complaints when you don't use commit synchronize. Again, it's not mandatory with GRES, but it's recommended as a best practice:

```
{master}[edit]
jnpr@R1-RE1# commit
warning: graceful-switchover is enabled, commit synchronize should be used
commit complete

{master}[edit]
jnpr@R1-RE1# commit synchronize
re1:
configuration check succeeds
re0:
commit complete
re1:
commit complete
```

You can avoid this nag by setting `commit synchronize` as a default, which is a feature that is mandatory for NSR:

```
jnpr@R1-RE1# set system commit synchronize

{master}[edit]
jnpr@R1-RE1# commit
re1:
configuration check succeeds
re0:
commit complete
re1:
commit complete
```

GRES itself does not mandate synchronized configurations. There can be specific reasons as to why you want to have different configurations between the two REs. It should be obvious that pronounced differences can impact on the relative success of a GRES event, so if you have no specific need for a different configuration it's best practice to use `commit synchronize` to ensure the current active configuration is mirrored to the BU RE, thus avoiding surprises at some future switchover, perhaps long after the configuration was modified but not synchronized.

GRES Options

The GRES feature has a few configuration options that add additional failover triggers. This section examines the various mechanisms that can trigger a GRES.

Disk Fail. You can configure whether a switchover should occur upon detection of a disk failure using the `on-disk-failure` statement at the `[edit chassis redundancy failover]` hierarchy:

```
jnpr@R1-RE1# set chassis redundancy failover ?
Possible completions:
+ apply-groups         Groups from which to inherit configuration data
+ apply-groups-except  Don't inherit configuration data from these groups
```

```
  on-disk-failure      Failover on disk failure
  on-loss-of-keepalives  Failover on loss of keepalive
```

> The RE has its own configuration for actions to be taken upon a hard disk failure. These include reboot or halt. You should not try and configure both actions for the same hard disk fail. When GRES is in effect, you should use `set chassis redundancy failover on-disk-failure`. Otherwise, use the `set chassis routing-engine on-disk-failure disk-failure-action [reset | halt]` statement when GRES is off. Note that having the RE with the disk problem perform a shutdown will trigger a GRES (if configured), given that keepalives will stop, but this method adds delay over the more direct approach of using the `set chassis redundancy failover on-disk-failure` statement.

Storage Media Failures

The failure of storage media is handled differently based on whether the primary of alternate media fails, and where the failure occurs on the master or backup RE:

- If the primary media on the master RE fails, the master reboots, and the backup assumes mastership. The level of service interruption is dependent on which HA features (GRES, NSR, GR, etc.) are enabled. The old master will attempt to restart from the alternate media, and if successful, will come back online as the backup RE. It will not become the master unless the new master fails or a manual switch is requested by the operator.
- If the alternate media on the master RE fails, the master will remain online and continue to operate as master unless `set chassis redundancy failover on-disk-failure` option is applied to the configuration. If this option is configured, the backup will assume mastership, and the old master will reboot. As before, the level of service interruption is dependent on which HA features are enabled. If the old master reboots successfully, it will come back online as the backup RE and will not become the master unless the new master fails or a manual switch is requested by the operator.
- If any media on the backup RE fails, the backup RE will reboot. If it boots successfully, it will remain the backup RE and will not become the master unless the master fails or a manual switch is requested by the operator.

Process Failure Induced Switchovers. You can also configure whether a switchover should occur upon detection of thrashing software processes at the `[edit system processes]` hierarchy. This configuration triggers a GRES if the related process, `rpd` in this case, is found to be thrashing, which is to say the daemon has started and stopped several times over a short interval (two or more times in approximately five seconds):

```
jnpr@R1-RE1# show system processes
routing failover other-routing-engine;
```

The effect of this setting is demonstrated by restarting the routing daemon a few times:

```
{master}[edit]
jnpr@R1-RE0# run restart routing immediately
error: Routing protocols process is not running
Routing protocols process started, pid 2236

{master}[edit]
jnpr@R1-RE0# run restart routing immediately

{master}[edit]
jnpr@R1-RE0# run restart routing immediately
error: Routing protocols process is not running
```

On the last restart attempt, an error is returned, indicating that the RPD process is no longer running, indicating it was not restarted due to thrashing. Also, note that after the previous process restart the local master has switched to the BU role:

```
{backup}[edit]
jnpr@R1-RE0# run restart routing immediately
error: Routing protocols process is not running
```

Verify GRES Operation

One you configure GRES, you want to make sure that after a commit synchronize both REs reflect either a master or BU status. In this section, the following GRES baseline is used:

```
{master}[edit]
jnpr@R1-RE0# show chassis
redundancy {
    graceful-switchover;
}
```

Things start with confirmation of a master and BU prompt on the two routing engines. There should never be two masters or two slaves. The prompt is confirmed to change for RE1 at R1, which is now in a backup role.

```
{backup}[edit]
jnpr@R1-RE1#
```

Next, you confirm that the BU RE is running the kernel synchronization daemon `ksyncd`:

```
{backup}
jnpr@R1-RE1>show system processes | match ksyncd
 5022  ??  S      0:00.15 /usr/sbin/ksyncd -N
 5034  ??  S      0:00.19 /usr/sbin/clksyncd -N
```

The output also shows the `clksyncd` daemon, responsible for precision time synchronization over Ethernet to support Synchronous Ethernet and other mobile backhaul technologies. With all looking good, the final indication that GRES is operation comes from a `show system switchover` command. This command is only valid on the BU RE, as it is the one doing all the synchronizing from the master:

```
{backup}
jnpr@R1-RE1>show system switchover
```

```
Graceful switchover: On
Configuration database: Ready
Kernel database: Ready
Peer state: Steady State
```

The output confirms that graceful switchover is on, that the configuration and kernel databases are currently synchronized, and that IPC connection to the master RE kernel is stable. This output indicates the system is ready to perform a GRES. You can get the master's RE view of the synchronization process with the `show database-replication` command:

```
{master}[edit]
jnpr@R1-RE0# run show database-replication ?
Possible completions:
  statistics           Show database replication statistics
  summary              Show database replication summary
{master}[edit]
jnpr@R1-RE0# run show database-replication

{master}[edit]
jnpr@R1-RE0# run show database-replication summary

General:
    Graceful Restart           Enabled
    Mastership                 Master
    Connection                 Up
    Database                   Synchronized
    Message Queue              Ready

{master}[edit]
jnpr@R1-RE0# run show database-replication statistics

General:
    Dropped connections        2
    Max buffer count           3
Message received:
    Size (bytes)               10320
    Processed                  162
Message sent:
    Size (bytes)               11805507
    Processed                  263
Message queue:
    Queue full                 0
    Max queue size             144032
```

Use the CLI's **`restart kernel-replication`** command to restart the `ksyncd` daemon on the current BU RE if it displays an error or is failing to complete synchronization in a reasonable period of time, which can vary according to scale but should not exceed 10 minutes. If the condition persists, you should confirm matched software versions on both REs, which is always a good idea when using GRES anyway.

While not strictly necessary, you always have your best chances of a GRES (or NSR) success when you have matched software versions on both REs. The exception is ISSU, as discussed in a later section.

If a persistent replication error is found even with matched versions, you may consider enabling ksyncd tracing, which is currently hidden and the only known use for the [edit system kernel-replication] hierarchy; as a hidden command, the results are undocumented and use is suggested only under guidance from JTAC:

```
jnpr@R1-RE0# set system kernel-replication ?
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
  |                    Pipe through a command
{master}[edit]
jnpr@R1-RE0# set system kernel-replication
```

As a hidden command, you have to type out traceoptions in its entirety, at which point help is again provided.

```
{master}[edit system kernel-replication]
jnpr@R1-RE0# set traceoptions flag ?
Possible completions:
  all                  Trace all events
  asp                  Trace ASP configuration events
  bd                   Trace bridge domain events
  config               Trace UI events and configuration changes
  cos                  Trace Class of Service events
  eventhandler         Trace event handler events
  firewall             Trace firewall events
  ifbd                 Trace ifbd events
  interface            Trace interface events
  ipc                  Trace IPC events
  monitor              Trace monitor events
  nexthop              Trace next-hop database events
  pfe                  Trace Packet Forwarding Engine events
  pic                  Trace PIC state events
  route                Trace route events
  rtsock               Trace routing socket events
  sample               Trace sample events
  stp                  Trace spanning tree protocol events
  sysconf              Trace system configurables events
{master}[edit system kernel-replication]
jnpr@R1-RE0# set traceoptions flag
```

A sample GRES trace file is shown, note the nondefault severity level in effect:

```
{master}[edit]
jnpr@R1-RE0# show system kernel-replication
traceoptions {
    level detail;
    flag stp;
    flag route;
    flag pfe;
    flag interface;
    flag bd;
}
```

With these settings, the interfaces stanza on the master RE is deactivated:

```
{master}[edit]
jnpr@R1-RE0# deactivate interfaces

{master}[edit]
jnpr@R1-RE0# commit
re0:
configuration check succeeds
. . .
And the folowing ksyncd trace is observed on the BU RE:
{backup}[edit]
jnpr@R1-RE1# Feb 22 10:47:57 write: op change ksync cookie seq
  0x10016, cookie 0xc8a3fd80:
Feb 22 10:47:57 send: slave ack cookie 0xc8a409c0 seqno 0x39 flags
  0x1 cookie64 0xc8a409c0
Feb 22 10:47:57 send: slave ack cookie 0xc8a409c0 seqno 0x3a flags
  0x1 cookie64 0xc8a409c0
Feb 22 10:47:57 send: slave ack cookie 0xc8a409c0 seqno 0x3b flags
  0x1 cookie64 0xc8a409c0
Feb 22 10:47:57 send: slave ack cookie 0xc8a3fd80 seqno 0x10016 flags
  0x3 cookie64 0xc8a3fd80
Feb 22 10:48:01 write: op change ksync cookie seq 0x10017, cookie
  0xc8a3fd80:
Feb 22 10:48:01 send: slave ack cookie 0xc8a409c0 seqno 0x3c flags
  0x1 cookie64 0xc8a409c0
Feb 22 10:48:01 send: slave ack cookie 0xc8a409c0 seqno 0x3d flags
  0x1 cookie64 0xc8a409c0
Feb 22 10:48:01 send: slave ack cookie 0xc8a409c0 seqno 0x3e flags
  0x1 cookie64 0xc8a409c0
Feb 22 10:48:01 send: slave ack cookie 0xc8a3fd80 seqno 0x10017 flags
  0x3 cookie64 0xc8a3fd80
Feb 22 10:48:02 write: op change ksync cookie seq 0x10018, cookie
  0xc8a3fd80:
Feb 22 10:48:02 send: slave ack cookie 0xc8a409c0 seqno 0x3f flags
  0x1 cookie64 0xc8a409c0
Feb 22 10:48:02 send: slave ack cookie 0xc8a409c0 seqno 0x40 flags
  0x1 cookie64 0xc8a409c0
Feb 22 10:48:02 send: slave ack cookie 0xc8a409c0 seqno 0x41 flags
  0x1 cookie64 0xc8a409c0
Feb 22 10:48:02 send: slave ack cookie 0xc8a3fd80 seqno 0x10018 flags
  0x3 cookie64 0xc8a3fd80
Feb 22 10:48:03                           output_queue : 0x00000000
Feb 22 10:48:03                         rewrite.plp_Q1 : 0x00000000
Feb 22 10:48:03                         rewrite.plp_Q2 : 0x00000000
Feb 22 10:48:03                         rewrite.plp_Q3 : 0x00000000
Feb 22 10:48:03                         rewrite.plp_Q4 : 0x00000000
Feb 22 10:48:03                      rewrite.no_plp_Q1 : 0x00000000
Feb 22 10:48:03                      rewrite.no_plp_Q2 : 0x00000000
Feb 22 10:48:03                      rewrite.no_plp_Q3 : 0x00000000
Feb 22 10:48:03                      rewrite.no_plp_Q4 : 0x00000000
Feb 22 10:48:03              IFTLV_TYPE_ID_INDEX_TUPLE :
Feb 22 10:48:03                                   type : 0x00000006
Feb 22 10:48:03                                     id : 0x0000001a
Feb 22 10:48:03                                    idx : 0x00000000
Feb 22 10:48:03                                   type : 0x00000006
Feb 22 10:48:03                                     id : 0x0000001b
Feb 22 10:48:03                                    idx : 0x00000000
Feb 22 10:48:03                                   type : 0x00000006
Feb 22 10:48:03                                     id : 0x0000001c
Feb 22 10:48:03                                    idx : 0x00000000
Feb 22 10:48:03 write: op change ifl irb unit 100 idx 329:
Feb 22 10:48:03                           output_queue : 0x00000000
Feb 22 10:48:03                         rewrite.plp_Q1 : 0x00000000
Feb 22 10:48:03                         rewrite.plp_Q2 : 0x00000000
Feb 22 10:48:03                         rewrite.plp_Q3 : 0x00000000
Feb 22 10:48:03                         rewrite.plp_Q4 : 0x00000000
Feb 22 10:48:03                      rewrite.no_plp_Q1 : 0x00000000
Feb 22 10:48:03                      rewrite.no_plp_Q2 : 0x00000000
Feb 22 10:48:03                      rewrite.no_plp_Q3 : 0x00000000
Feb 22 10:48:03                      rewrite.no_plp_Q4 : 0x00000000
Feb 22 10:48:03              IFTLV_TYPE_ID_INDEX_TUPLE :
Feb 22 10:48:03                                   type : 0x00000006
Feb 22 10:48:03                                     id : 0x0000001a
Feb 22 10:48:03                                    idx : 0x00000000
Feb 22 10:48:03                                   type : 0x00000006
Feb 22 10:48:03                                     id : 0x0000001b
Feb 22 10:48:03                                    idx : 0x00000000
Feb 22 10:48:03                                   type : 0x00000006
Feb 22 10:48:03                                     id : 0x0000001c
Feb 22 10:48:03                                    idx : 0x00000000
Feb 22 10:48:03 write: op change ifl irb unit 200 idx 330:
Feb 22 10:48:03 write op delete route prefix 224.0.0.18 nhidx 608:
Feb 22 10:48:03                           output_queue : 0x00000000
Feb 22 10:48:03                         rewrite.plp_Q1 : 0x00000000
Feb 22 10:48:03                         rewrite.plp_Q2 : 0x00000000
Feb 22 10:48:03                         rewrite.plp_Q3 : 0x00000000
Feb 22 10:48:03                         rewrite.plp_Q4 : 0x00000000
Feb 22 10:48:03                      rewrite.no_plp_Q1 : 0x00000000
Feb 22 10:48:03                      rewrite.no_plp_Q2 : 0x00000000
Feb 22 10:48:03                      rewrite.no_plp_Q3 : 0x00000000
Feb 22 10:48:03                      rewrite.no_plp_Q4 : 0x00000000
Feb 22 10:48:03              IFTLV_TYPE_ID_I
. . .
Feb 22 10:48:04 write op delete route prefix 120.120.35.0 rttype user
  nhidx 1048575 nhtype indr:
Feb 22 10:48:04 send: slave ack cookie 0xc8a409c0 seqno 0x4a flags 0x1
  cookie64 0xc8a409c0
Feb 22 10:48:04 write op delete route prefix 120.120.34.0 rttype user
  nhidx 1048575 nhtype indr:
Feb 22 10:48:04 write op delete route prefix 120.120.33.0 rttype user
  nhidx 1048575 nhtype indr:
Feb 22 10:48:04 write op delete route prefix 120.120.32.0 rttype user
  nhidx 1048575 nhtype indr:
Feb 22 10:48:04 write op delete route prefix 120.120.31.0 rttype user
```

```
  nhidx 1048575 nhtype indr:
Feb 22 10:48:04 write op delete route prefix 120.120.30.0 rttype user
  nhidx 1048575 nhtype indr:
Feb 22 10:48:04 write op delete route prefix 120.120.29.0 rttype user
  nhidx 1048575 nhtype indr:
Feb 22 10:48:04 write op delete route prefix 120.120.28.0 rttype user
  nhidx 1048575 nhtype indr:
Feb 22 10:48:04 write op delete route prefix 120.120.27.0 rttype user
  nhidx 1048575 nhtype indr:
Feb 22 10:48:04 write op delete route prefix 120.120.26.0 rttype user
  nhidx 1048575 nhtype indr:
. . .
```

Be sure to remove any tracing you have added when it's no longer needed. At scale, the additional burden of tracing kernel replication can lead to long delays in completing replication.

GRES, Before and After. To finish this section, we provide a quick demonstration of the net payoff you get with GRES. Things begin with disabling of GRES at R1, where RE0 is the current master:

```
{master}[edit]
jnpr@R1-RE0# delete chassis redundancy graceful-switchover

{master}[edit]
jnpr@R1-RE0# commit
re0:
configuration check succeeds
re1:
commit complete
re0:
commit complete

[edit]
jnpr@R1-RE0#
```

Note again how after disabling GRES, the CLI banner no longer displays a master/backup designation.

> Even when the banner does not display master or backup, you can always tell which RE is master with a **`show chassis hardware`** command, as only the master can access chassis information. Alternatively, the **`show chassis routing-engine`** command also displays mastership state.

With GRES off, you confirm that all FPCs are up and that a given interface used in the test topology, in this case the xe-2/1/1 interface, is configured and operational:

```
jnpr@R1-RE0# run show interfaces xe-2/1/1
Physical interface: xe-2/1/1, Enabled, Physical link is Up
  Interface index: 199, SNMP ifIndex: 5516
  Link-level type: Ethernet, MTU: 1514, LAN-PHY mode, Speed: 10Gbps,
    Loopback: None, Source filtering: Disabled,
  Flow control: Enabled
  Device flags   : Present Running
  Interface flags: SNMP-Traps Internal: 0x4000
  Link flags     : None
  CoS queues     : 8 supported, 8 maximum usable queues
  Schedulers     : 0
  Current address: 00:1f:12:b8:8d:d0, Hardware address: 00:1f:12:b8:8d:d0
. . .

[edit]
jnpr@R1-RE0# run show chassis fpc
                     Temp  CPU Utilization (%)   Memory    Utilization (%)
Slot State            (C)  Total  Interrupt      DRAM (MB) Heap     Buffer
  0  Empty
  1  Online            40     21          0       2048       12         13
  2  Online            38     24          0       2048       11         13
```

Also, at this time, you issue a **show switchover** on the BU RE to confirm GRES is off:

```
jnpr@R1-RE1# run show system switchover
Graceful switchover: Off
Peer state: Steady State
```

And now, you perform an Ungraceful routing engine switchover (UGRES):

```
[edit]
jnpr@R1-RE1# run request chassis routing-engine master acquire no-confirm
Resolving mastership...
Complete. The local routing engine becomes the master.
```

Immediately after the switch, you confirm that the various chassis components have been reset; this is expected: given the lack of kernel synchronization the new master has no alternative but to start fresh to ensure internal consistency between the control and dataplane.

```
[edit]
jnpr@R1-RE1# run show chassis fpc
                     Temp  CPU Utilization (%)   Memory    Utilization (%)
Slot State            (C)  Total  Interrupt      DRAM (MB) Heap     Buffer
  0  Empty
  1  Present          Testing
  2  Present          Testing

[edit]
jnpr@R1-RE1# run show interfaces xe-2/1/1
error: device xe-2/1/1 not found
```

You have to admit that was most ungraceful. To show the contrast, GRES is again enabled (recall that commit synchronize has been left in place), and the change is committed:

```
[edit]
jnpr@R1-RE0# set chassis redundancy graceful-switchover

[edit]
jnpr@R1-RE0# commit
re0:
```

```
configuration check succeeds
re1:
commit complete
re0:
commit complete
```

Before performing another switchover, synchronization is confirmed on the backup RE:

```
[edit]
jnpr@R1-RE1# run show system switchover
Graceful switchover: On
Configuration database: Ready
Kernel database: Ready
Peer state: Steady State
```

The synchronization state is good, so you quickly confirm PFE state as in the non-GRES switchover case:

```
{master}[edit]
jnpr@R1-RE0# run show chassis fpc
                     Temp  CPU Utilization (%)   Memory    Utilization (%)
Slot State            (C)  Total  Interrupt      DRAM (MB) Heap     Buffer
  0  Empty
  1  Online            40     21         0       2048       12          13
  2  Online            39     23         0       2048       11          13

{master}[edit]
jnpr@R1-RE0# run show interfaces xe-2/1/1 terse
Interface               Admin Link Proto    Local                 Remote
xe-2/1/1                up    up
xe-2/1/1.0              up    up   inet     192.168.0.2/30
                                   multiservice
```

And now, a graceful switchover is performed. The CLI timestamp function is evoked first (not shown) to help give a sense of the time base in which the various commands were executed:

```
{backup}[edit]
jnpr@R1-RE1# run request chassis routing-engine master acquire no-confirm
Feb 01 11:21:45
Resolving mastership...
Complete. The local routing engine becomes the master.

{master}[edit]
jnpr@R1-RE1# run show chassis fpc
Feb 01 11:21:56
                     Temp  CPU Utilization (%)   Memory    Utilization (%)
Slot State            (C)  Total  Interrupt      DRAM (MB) Heap     Buffer
  0  Empty
  1  Online           Testing  27         0       2048       12          13
  2  Online           Testing  27         0       2048       11          13
```

As expected, the new master has not reset any FPCs, though it does have to probe them for current state such as temperature. The test interface also remains and continues to use its preswitchover configuration. This is both due to lack of reset and because the two REs had the same configuration as a result of using `commit synchronize`.

```
{master}[edit]
jnpr@R1-RE1# run show interfaces xe-2/1/1 terse
Feb 01 11:22:03
Interface               Admin Link Proto    Local                 Remote
xe-2/1/1                up    up
xe-2/1/1.0              up    up   inet     192.168.0.2/30
                                   multiservice
```

A bit later, the chassis component state is updated in the new master:

```
{master}[edit]
jnpr@R1-RE1# run show chassis fpc
Feb 01 11:22:12
                     Temp  CPU Utilization (%)   Memory    Utilization (%)
Slot State            (C)  Total  Interrupt      DRAM (MB) Heap     Buffer
  0  Empty
  1  Online            39      23          0       2048       12         13
  2  Online            38      29          0       2048       11         13
```

The results confirm a successful GRES event. While the FPCs and interfaces persisted through the switchover, it must be stressed that none of the control plane protocols did. Any BGP, OSPF, ISIS, LDP, STP, etc., sessions were reset and then reestablished on the new master. Also, as Graceful Restart (GR) is not in effect, peering routers will immediately begin removing any FIB entries that relate to the reset sessions, which means that traffic forwarding stops. Even so, not having to wait for FPC reboot and interface initialization means that recover will be faster than in the non GRES case.

You must run peer-to-peer periodic protocols such as BFD and LACP in distributed mode (the default) to ensure sessions do not flap at GRES by having their periodic hello needs handled by the `ppmd` process in the PFE itself. Note that even when running in distributed mode, certain sessions, such as OSPF, multi-hop BFD, or IPv6 link local–based protocols such as OSPF3, remain RE based in the 11.4 release, and therefore need to run relatively long timers to ensure the sessions remain up through a GRES event. For multi-hop BFD, the minimum interval should be at least 2,500 ms.

GRES and Software Upgrade/Downgrades

When testing, the author routinely upgrades or downgrades both REs at the same time, while NSR/GRES is in effect via the `force` and `no-validate` switches to the `request system software add` command. However, this approach is not officially supported, and tends to increase network disruption as both REs go offline at nearly the same time to load the new software, leaving the router inoperable for 15 minutes or so.

If you omit the `no-validate` switch, a software installation aborts when GRES is found to be in effect:

```
Using jservices-crypto-11.4R1.9.tgz
Hardware Database regeneration succeeded
Validating against /config/juniper.conf.gz
Chassis control process: [edit chassis redundancy]
Chassis control process:   'graceful-switchover'
Chassis control process:     Graceful switchover configured!
mgd: error: configuration check-out failed
Validation failed
WARNING: Current configuration not compatible with /var/home/jnpr/
  jinstall-11.4R1.9-domestic-signed.tgz
```

The official upgrade (or downgrade) procedure for 11.4 when GRES is in effect is documented at *http://www.juniper.net/techpubs/en_US/junos11.4/information-products/topic-collections/software-installation-and-upgrade-guide/swconfig-install.pdf#search=%22Junos%20OS%20Installation%20and%20Upgrade%20Guide.%22*.

The summary is as following:

1. Disable GRES (and NSR if enabled), commit and synchronize the changes to both REs.
2. Upgrade (or downgrade) the current BU RE.
3. After completion, and when all appears to be OK, switch control to the former BU RE, which becomes the new master. Note this is an ungraceful switch given that GRES is off. There is a hit to both control and data plane as the PFE is rebooted, etc.
4. Upgrade (or downgrade) the new BU/old master.
5. After completion, and when all appears to be OK, restore the original GRES (and potentially NSR) config with a rollback 1 on the new master RE. This puts GRES (and NSR if so configured) back into effect and leaves the system on RE1 as the current master.
6. If desired, you can perform a graceful switchover to make RE0 the new master. If NSR or GR is in effect, this switchover can bit hitless to the control and dataplane, or to just the dataplane, respectively.

GRES Summary

GRES can be used as a standalone feature on any Junos router with redundant REs. In most cases, GRES is used as the building block for additional HA features such as GR or NSR. The next section builds upon the GRES foundation by adding GR to provide dataplane resiliency through a switchover.

Graceful Restart

Graceful Restart (GR) is also referred to as Nonstop Forwarding (NSF) and describes a router's ability to maintaining forwarding state through a protocol-level restart or GRES event, leveraging the fact that modern routers use a separated control and dataplane, which in turn allows decoupling such that a restart of one no longer forces the restart of the other.

A protocol restart can occur due to intentional or unintentional reasons. For example, an operator choosing to restart the routing process, rebooting the router, or upgrading its software are examples of the former, whereas a routing process crash or hardware-induced RE switchover fall into the latter category.

GR is not a panacea of guaranteed success, and as mentioned previously the trend is to move to Nonstop Routing as support for the protocol you need in your network becomes available. One upside to GR is that it can be used on routers with a single RE; both GRES and NSR require redundant REs to work.

GR Shortcomings

The primary drawback to GR is the need for protocol extensions and helper support in neighboring routers, and a stable network topology. If any neighbors do not support GR, or if other changes occur in the network, GR ends up aborting and loss results in the dataplane. Even when all goes to plan and the GR event succeeds, which means there is zero loss in the data plane, there is still control plane flap and a subsequent need for protocol reconvergence between the restarting router and its helpers, both of which are avoided with NSR.

Currently, there is no GR support for Layer 2 control protocols (i.e., STP). Currently, Layer 2 HA is available only as nonstop bridging (NSB), as discussed in a later section. Note this restriction extends to the BFD protocol, which should not be combined with GR, for reasons described in a later section.

Lastly, GR is predicated on the network being stable during the restart event, which can easily be several minutes long. During this time, if any other (relevant) reconvergence is detected by either the restarting or helper nodes, the GR process is aborted. This is because a loop-free topology can no longer be assumed through the restart when topology changes are occurring. Note that in this context, relevant refers to the receipt of a newly flooded LSA that requires processing and reflooding.

> Unlike NSR, GR itself does not require that dual REs be present or that GRES be enabled. However, GR works best with GRES to provide GR support in the event of an RE failure.

Graceful Restart Operation: OSPF

Many protocols support GR, and each have their own specifics as to how GR is implemented. However, as with most things in life, there are many general truths that apply to all things GR. This section describes GR in the context of OSPF, but does so in a manner that also exposes the reader to general GR terminology and working principals that apply to all supported protocols. OSPF GR is documented in RFC 3623 “Graceful OSPF Restart,” with modern implementations also using enhanced procedures defined in RFC 4812 "OSPF Restart Signaling," as described in the following. Figure 9-3 provides a sample topology to help ground the discussion.

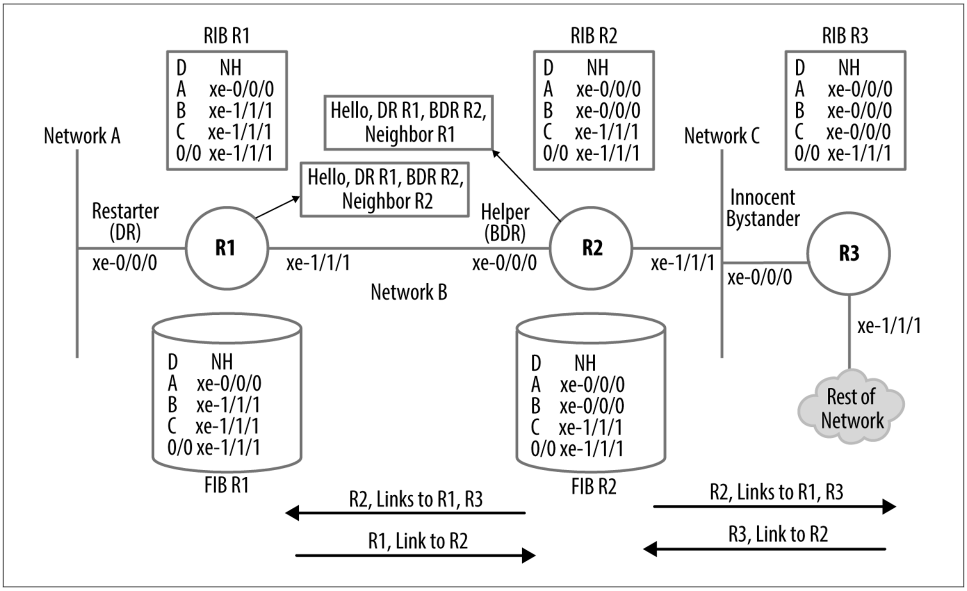


Figure 9-3. Sample OSPF Network for Graceful Restarting.

The figure shows a three router network with simplified Routing and Forwarding Information Bases (RIB/FIB), based on the networks and interfaces shown. While based on OSPF, other protocols perform GR in a similar manner. Things begin with a stable network with complete routing information in all nodes. Recall that in OSPF each node uses a router LSA to advertise its direct links, and to identify the link type as stub or transit (attached to a neighboring router). In addition, the OSPF hello packets sent on multiaccess networks indicate the current DR, BDR, and list all neighbors for which two-way communications has been seen. The figure shows the (greatly simplified) router LSAs at the bottom, and the hello packet from R2 near the top. Recall also that in OSPF hello packets are used by OSPF to dynamically locate other OSPF speakers, which can then lead to adjacency formation; adjacencies are not formed to all neighbors in OSPF, for example on a broadcast LAN where DR-Other routers only form full adjacencies to the Designated and Backup Designated Routers (DR/BDR).

In this example, R1 is the LAN segment's Designated Router (DR) and its neighbor R2 is the Backup DR (BDR). R2's hello packet is listing R1 as the DR, and as a neighbor, and the flooding of the type 1 LSAs has given all routers knowledge of the various subnets, as shown by the RIBs/FIBs shown for each router.

Before getting all GR with-it, let's start with some terminology and concepts.

Restarting Router

The restarting router, as its name implies, is the router that undergoes some form of protocol restart. This can be planned, as in the case of a maintenance action, or unplanned, as a result of a software crash. The procedure can vary for each, and a successful GR is never guaranteed.

In the case of OSPF, there is no a priori confirmation that a given neighbor is willing to perform the helper role. Nonetheless, at restart, or if not possible, after the restart, OSPF sends a Grace LSA to all neighbors informing them of the restart and the maximum period of time they should wait for the process to complete.

After a restart, the restarting router reforms its adjacencies but does not flood any LSAs; instead, it uses its helpers to download it with its pre-restart RIB state. In the case of OSPF, this means reflooding from the helper router to the restarting router the various LSAs that make up the area's LSDB. As a result, the restarting router receives copies of its own LSAs (all routers have the same LSDB, which includes their own self-originated LSAs, which also helps to ensure that new sequence numbers aren't generated for these LSAs) and uses them to determine which adjacencies it previously had so that it can determine when all pre-restart adjacencies have been reformed. On multiaccess segments, the restarting router uses the hello packets sent by its helpers to determine if it was the segment's DR, and if so, it recovers that functionality as well.

When all adjacencies have been reformed, and no reason for an abort has been found, GR is exited by purging the Grace LSA (restarting router refloods the Grace LSA with age set near max-age/3,600 seconds). Only now does the restarting router use its RIB to make any necessary updates to its FIB, which has remained frozen throughout the restart event. At the end of the restart, successful or not, both the restarting router and helper reflood their Router (Type 1) and Network (Type 2) LSAs, the latter being a function of whether one or the other is the segment's DR.

Grace LSA. The grace-LSA is a Type 9 Opaque LSA coded with an Opaque Type of 3 and an Opaque ID equal to 0. Opaque LSAs are used to allow OSPF to transport information that is not directly related to routing, or in some cases, not even intended for OSPF (i.e., a TED that's built by OSPF but used by CSPF in support of RSVP Traffic-Engineering [TE]). This grace-LSA has a link-local scope, so it travels only to the restarting routers' immediate adjacencies. The age of the LSA is set to 0 so that later the LSA age can be compared to the advertised restart duration to determine when the restart time has expired. The body of the LSA is TLV-coded, with TLVs defined for a restart reason as well as the maximum duration. For multipoint topologies, the restarting router also codes the IP address of the sending interface for identification purposes. Restart reasons include 0 (unknown), 1 (software restart), 2 (software reload/upgrade) or 3 (switch to redundant control processor).

Helper Router

In the case of OSPF, the role of the helper router is rather straightforward. That is, unless a reason for an abort is found, it continues to advertise LSA/hellos indicating an ongoing adjacency with the restarting router as if nothing ever happened.

This behavior is the heart and soul of OSPF GR. In normal operation, the adjacency will be lost during the restart and the neighbor will immediately remove the related routes from its FIB while flooding updated Type 1/2 LSAs reporting the loss of the restarting router. As a result, all other routers in the area also remove the restarting router from their SPF tree and rerun their SPF calculation in an attempt to route around the restarting router. Instead, when all goes to GR-plan, no FIB updates are made in the helping or restarting routers, and no LSA updates are flooded by the helping routers, which means the rest of the network continues to forward as it did before the restart, hence the term Nonstop Forwarding (NSF).

In most cases, helper mode and graceful restart are independent. You can disable graceful restart in the configuration but still allow the router to cooperate with a neighbor attempting to restart gracefully, or you can enable GR and then on a protocol basis chose to disable restart or helper mode as desired.

Aborting GR

GR can be aborted for many reasons. In all cases, the result is the same as if GR were not in effect, which is to say the restarting router goes missing from the OSPF database, LSAs are flooded through the area to report the change, FIBs are modified, and packets destined to the restarting router's direct connections begin hitting the floor. It's a bloody mess, but this is Earth, and as they say, "feces transpires."

> Combining BFD with GR is a great way to have GR abort, which in turn results in disruption to the data plane. You can combine BFD with NSR (or basic GRES) as described later.

In the case of OSPF, it's better to abort and route around the restarting node than to make assumptions that could lead to a forwarding loop. As such, any changes that are relevant to the topology during the restart event (i.e., a new LSA that needs to be flooded to the restarting router but cannot be because it's restarting at the moment) are all just cause for GR termination. Other reasons include a router that does not support or has been configured not to support the GR helper mode, or the expiration of the advertised grace period before the Grace LSA is purged by the restarting router.

A Graceful Restart, at Last

Having discussed all the theory, we can now revisit our sample OSPF network, now in the context of an OSPF restart event. The reader is warned this will be anticlimatic; an updated figure is provided in Figure 9-4.

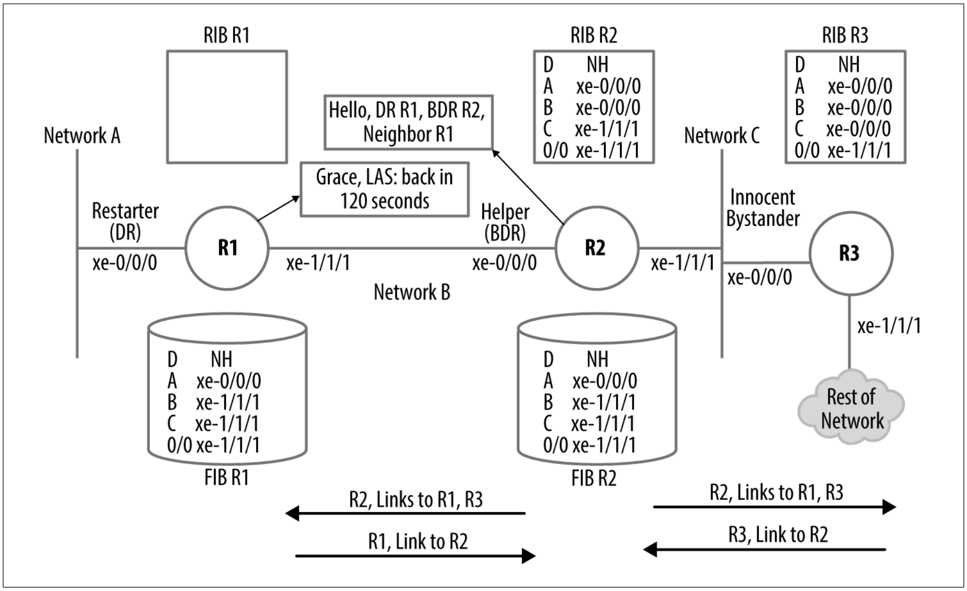


Figure 9-4. An OSPF GR Event.

If you get the feeling that not much changed, then you get the point of GR. The figure shows that R1, after sending its grace-LSA, has gone missing in the control plane. No hellos are being sent from R1 to R2. Yet R2, being the good helper that it is, has taken note of the grace-LSA, and for the remainder of the specified duration continues to send hellos listing R1 as a neighbor and DR; specifically, it does *not* flood any LSA reporting any change at the restarting router, so to remote router R3 nothing has changed at all. Note also how R1 has kept its pre-restart FIB intact, allowing transit traffic to be routed through it to reach network A, just as was the case in the pre-restart network. Assuming that R1 can reform its adjacencies with R2 in the specified period of time, all ends well with this story. If, on the other hand, R3 were to experience an OSPF interface outage, and as a result has to reflood a modified type 1 router LSA, then R2 is forced to abort the restart and begin admitting that R1 is also gone.

Unlike NSR, GR works through a restart of the routing process, such as when the operator issues a `restart routing` command. In fact, in plain old GRES/GR, where NSR is not enabled, `rpd` does not run on the BU RE. As a result, after a switchover the RPD process is restarted on the new master, which again is not a problem for GR. A later section details how NSR differs in this regard.

A Fly in the Ointment—And an Improved GR for OSPF

The original specification forPAGE RANGE W/ 1 SUB: OSPF GR, as laid out in RFC 3632, was a bit vague when it stated in section 2 that, "After the router restarts/reloads, it must change its OSPF processing somewhat until it re-establishes full adjacencies with all its former fully-adjacent neighbors." Seems simple enough, but to do this the restarting router has to send hello packets.

And therein lies the rub. Those hello packets may not list any neighbors, given the restarting router has, well, just restarted. As it turned out, some implementations of OSPF GR would abort a GR as part of the two-way connectivity check with the announcing neighbor (i.e., the router would generate a 1-way Received event for the neighbor if it does not find its own router ID in the list of neighbors, as described in s*http://tools.ietf.org/html/rfc2328#section-10.5*), which resulted in adjacency termination and, as a result, a most ungraceful restart.

OSPF Restart Signaling RFCs 4811, 4812, and 4813. As it happens, independent work was also being done on a way to provide OSPF with a type of Out of Band (OoB) communications channel that could be used for a variety of purposes. Known as OSPF Link Local Signaling (LLS), the mechanism was first defined in RFC 4813, which was later obsoleted by RFC 5613. Rather than define a new OSPF packet type, the choice was made to make use of TLV extensions to OSPF hello and Database Description (DD) packets to convey arbitrary information. The data included in the LLS block attached to a hello packet may be used for dynamic signaling given that hello packets may be sent at any time, albeit without any delivery guarantees. In contrast, data sent with DD packets is guaranteed to be delivered as part of the adjacency formation process. A new OSPF LLS options bit, referred to as the "L-bit", is set to indicate whether a given hello or DD packet contains an LLS data block. In other words, the LLS data block is only examined if the L-bit is set.

While LLS was being defined in 4813 (now 5613), work was also under way to define a mechanism for OSPF to resynchronize its LSDB without forcing a transition to the exchange-start state. Once again, such a transition could force an abort of a graceful restart. This work is defined in RFC 4811 "OSPF Out-of-Band LinkState Database (LSDB) Resynchronization" and makes use of OSPF DD packets with LLS TLVs attached. Specifically, a new LR bit (LSDB Resynchronization) is defined for use in LLS Extended Options TLV. Routers set the LR-bit to announce OOB LSDB resynchronization capability in both hello and DBD packets.

Of primary interest in this discussion is RFC 4812 "OSPF Restart Signaling," which conveniently sits (numerically) between the LLS and LSDB resynchronization RFCs just discussed. RFC 4812 defines a Restart-Signal (RS) bit conveyed in the Extended Options (EO) TLV in the Link-Local Signaling (LLS) block of hello packets sent by the restarting router. Upon reception of the RS option, the helpers skip the two-way connectivity check, thereby solving the issue of premature termination due to a helper receiving hellos with an empty neighbor list. In addition, RFC 4812 specifies use of the LLS-based method of LSDB resynchronization, as specified in RFC 4811.

RFC 4812 introduces two new fields in the neighbor data structure: the RestartState flag and ResyncTimeout timer. The RestartState flag indicates that a hello packet with the RS-bit set has been received and that the local router expects its neighbor to go through the LSDB resynchronization procedure specified in RFC 4813/5613 (using LLS). When that is the case, the ResyncTimeout timer is used to determine how long the helper will wait for the LLS based LSDB resynch to begin before it declares one-way state, aborting the restart.

After a restart, an RFC 4812-compliant router sets the RS-bit in the EO-TLV of their hello packets when it's not sure that all neighbors are listed in the hello packet but the restarting router wants them to preserve their adjacencies anyway. When an OSPF router receives a hello packet that contains the LLS block with the EO-TLV that has the RS-bit set, the router should skip the two-way connectivity check, as mentioned previously. The helper should also send a unicast hello back to the restarting router to speed up learning of previously known neighbors. These unicast hello packets don't have the RS-bit set.

Graceful Restart and other Routing Protocols

Junos offers GR support for virtually all routing protocols. While complete coverage of all things GR is beyond the scope of this chapter, a brief summary is provided for each major protocol.

BGP

When a router enabled for BGP graceful restart restarts, it retains BGP peer routes in its forwarding table and marks them as stale. However, it continues to forward traffic to other peers (or receiving peers) during the restart. To reestablish sessions, the restarting router sets the "restart state" bit in the BGP OPEN message and sends it to all participating peers. The receiving peers reply to the restarting router with messages containing end-of-routing-table markers. When the restarting router receives all replies from the receiving peers, the restarting router performs route selection, the forwarding table is updated, and the routes previously marked as stale are discarded. At this point, all BGP sessions are reestablished and the restarting peer can receive and process BGP messages as usual.

While the restarting router does its processing, the receiving peers also temporarily retain routing information. When a receiving peer detects a TCP transport reset, it retains the routes received and marks the routes as stale. After the session is reestablished with the restarting router, the stale routes are replaced with updated route information.

Restart procedures for BGP are defined in RFC 4724 "Graceful Restart Mechanism for BGP."

ES-IS

When graceful restart for ES-IS is enabled, the routes-to-end systems or intermediate systems are not removed from the forwarding table. The adjacencies are reestablished after restart is complete. Note: ES-IS is supported only on the J-Series

Services Router as well as SRX Branch platforms and starting with release v11.2, Trio-based MX platforms.

IS-IS

Normally, IS-IS routers move neighbor adjacencies to the down state when changes occur. However, a router enabled for IS-IS graceful restart sends out hello messages with the Restart Request (RR) bit set in a restart type length value (TLV) message. This indicates to neighboring routers that a graceful restart is in progress and to leave the IS-IS adjacency intact. Besides maintaining the adjacency, the neighbors send complete sequence number PDUs (CSNPs) to the restarting router and flood their entire database.

The restarting router never floods any of its own link-state PDUs (LSPs), including pseudonode LSPs, to IS-IS neighbors while undergoing graceful restart. This enables neighbors to reestablish their adjacencies without transitioning to the down state and enables the restarting router to reinitiate database synchronization.

IS-IS restart mechanisms are defined in RFC 5306 “Restart Signaling for IS-IS.”

OSPF and OSPFv3

While the focus of the previous discussion on general GR mechanism, a review never hurts. When a router enabled for OSPF graceful restart restarts, it retains routes learned before the restart in its forwarding table. The router does not allow new OSPF link-state advertisements (LSAs) to update the routing table.

To begin the process, the restarting router sends a grace LSA to all neighbors. In response, the helper routers enter helper mode and send an acknowledgement back to the restarting router. If there are no topology changes, the helper routers continue to advertise LSAs as if the restarting router had remained in continuous OSPF operation.

When the restarting router reforms adjacencies with all its pre-restart helper routers, it resumes normal operation and begins selecting routes and performing updates to the forwarding table. The restart ends when the Grace LSA is flushed, the restart timer expires, or the process aborts because topology change is detected.

Junos supports both standard and restart signaling-based helper modes, and both are enabled by default whether or not GR is enabled globally. Currently, restart signaling-based graceful restart helper mode is not supported for OSPFv3 configurations.

PIM Sparse Mode

PIM sparse mode uses a mechanism called a generation identifier to indicate the need for graceful restart. Generation identifiers are included by default in PIM hello messages. An initial generation identifier is created by each PIM neighbor to establish device capabilities. When one of the PIM neighbors restarts, it sends a new generation identifier to its neighbors. All neighbors that support graceful restart and are connected by point-to-point links assist by sending multicast updates to the restarting neighbor.

The restart phase completes when either the PIM state becomes stable or when the restart interval timer expires. If the neighbors do not support graceful restart or connect to each other using multipoint interfaces, the restarting router uses the restart interval timer to define the restart period.

RIP and RIPng

There is no restart specification for RIP as its built-in, so to speak. When a router enabled for RIP graceful restart restarts, routes that have been installed in the FIB are simply not deleted. Because no helper router assists in the restart, these routes are retained in the forwarding table while the router restarts (rather than being discarded or refreshed).

RSVP

RSVP graceful restart is described in RFC 3473 "Generalized Multi-Protocol Label Switching (GMPLS) Signaling Resource ReserVation Protocol-Traffic Engineering (RSVP-TE) Extensions" (only Section 9, "Fault Handling"). For the restarting router, RSVP graceful restart attempts to maintain the routes installed by RSVP and the allocated labels, so that traffic continues to be forwarded without disruption. RSVP graceful restart is done quickly enough to reduce or eliminate the impact on neighboring nodes. The neighboring routers must have RSVP graceful restart helper mode enabled, thus allowing them to assist a router attempting to restart RSVP.

An object called Restart Cap is sent in RSVP hello messages to advertise a node's restart capability. The neighboring node sends a Recover Label object to the restarting node to recover its forwarding state. This object is essentially the old label that the restarting node advertised before the node restarted. The following assumptions are made about a neighbor based on the Restart Cap object:

A neighbor that does not advertise the Restart Cap object in its hello messages cannot assist a router with state or label recovery, nor can it perform an RSVP graceful restart.

After a restart, a neighbor advertising a Restart Cap object with a restart time equal to any value and a recovery time equal to 0 has not preserved its forwarding state. When a recovery time equals 0, the neighbor is considered dead and any states related to this neighbor are purged, regardless of the value of the restart time.

After a restart, a neighbor advertising its recovery time with a value other than 0 can keep or has kept the forwarding state. If the local router is helping its neighbor with restart or recovery procedures, it sends a Recover Label object to this neighbor.

LDP

During session initialization, a router advertises its ability to perform LDP graceful restart or to take advantage of a neighbor performing LDP graceful restart by sending the graceful restart TLV. This TLV contains two fields relevant to LDP graceful restart: the reconnect time and the recovery time. The values of the reconnect and recovery times indicate the graceful restart capabilities supported by the router.

When a router discovers that a neighboring router is restarting, it waits until the end of the recovery time before attempting to reconnect. The recovery time is the length of time a router waits for LDP to restart gracefully. The recovery time period begins when an initialization message is sent or received. This time period is also typically the length of time that a neighboring router maintains its information about the restarting router, allowing it to continue to forward traffic.

The following are some of the behaviors associated with LDP graceful restart:

- Outgoing labels are not maintained in restarts. New outgoing labels are allocated.
- When a router is restarting, no label-map messages are sent to neighbors that support graceful restart until the restarting router has stabilized (label-map messages are immediately sent to neighbors that do not support graceful restart). However, all other messages (keepalive, address-message, notification, and release) are sent as usual. Distributing these other messages prevents the router from distributing incomplete information.
- Helper mode and graceful restart are independent. You can disable graceful restart in the configuration, but still allow the router to cooperate with a neighbor attempting to restart gracefully.
- In Junos, the defaults have graceful restart helper mode enabled while graceful restart is disabled. Thus, the default behavior of a router is to assist neighboring routers attempting a graceful restart, but not to attempt a graceful restart itself.
- LDP Graceful restart is defined in RFC 3478, “Graceful Restart Mechanism for Label Distribution Protocol.”

Junos GR Support by Release

Junos offers restart support for virtually all protocols, and starting as far back as release v5.3, GR is clearly a rather mature technology. GR support and version requirements as of this writing are:

Release v5.3 for aggregate route, BGP, IS-IS, OSPF, RIP, RIPng, or static routes
Release v5.5 for RSVP on egress provider edge (PE) routers
Release v5.5 for LDP graceful restart
Release v5.6 for the CCC, TCC, Layer 2 VPN, or Layer 3 VPN implementations of graceful restart
Release v6.1 for RSVP graceful restart on ingress PE routers
Release v6.4 for PIM sparse mode graceful restart
Release v7.4 for ES-IS graceful restart (J-Series Services Routers)
Release v8.5 for BFD session (helper mode only)—If a node is undergoing a graceful restart and its BFD sessions are distributed to the Packet Forwarding Engine, the peer node can help the peer with the graceful restart
Release v9.2 for BGP to support helper mode without requiring that graceful restart be configured

Release v11.3 Support for restart signaling-based helper mode for OSPF graceful restart

Configure and Verify OSPF GR

This section details GR configuration and operation for the OSPF protocol in Junos, but the concepts and operation are similar for all GR-supported protocols. Configuring GR in Junos is quite trivial; in fact, for most protocols, helper mode is enabled with no explicit GR configuration, which means you have to *explicitly disable* helper mode for those protocols when a complete absence of GR is required.

Enable Graceful-Restart Globally

To enable restart for all supported protocols in the main routing instance, all that is required is a single `set routing-options graceful-restart` statement. At that point, you can then configure various restart and helper mode attributes for each specific protocol as desired. You can also enable restart in routing instances, and logical systems, a point that is often overlooked on provider-edge (PE) routers, where you generally want HA in both the main and VRF instances. As shown in the following, there are very few options to the `graceful-restart` statement:

```
{master}[edit]
jnpr@R1-RE0# set routing-options graceful-restart ?
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
  disable              Disable graceful restart
  restart-duration     Maximum time for which router is in graceful restart (120..900)
  |                    Pipe through a command
```

Setting `disable` is the same as not enabling GR. Note again that this statement controls the local node's ability to perform a graceful restart; helper modes are generally enabled with no explicit configuration. The `restart-duration` parameter is somewhat significant. The value specified places a maximum limit on how long any restart event can last, so it's critical this value be longer than that used by any specific protocol. Note also that some protocols can only begin their restart as a result of a lower-level protocol completing its restart. For example, LDP restart is dependent on the underlying IGP, typically IS-IS or OSPF, completing its restart successfully.

As an example, the global default for restart duration is 300 seconds while the default restart duration for OSPF is only 180 seconds.

> You must ensure that the global restart duration is longer than that needed by any protocol or GR will abort. Note that some protocols are dependent upon others, so it's not always a simple case of setting global restart duration to be longer than the value used by any individual protocol. For scaled configuration, consider setting the value longer, but in general is best not to set a value less than the default.

OSPF GR Options

There's not a whole lot to configure as far as OSPF and GR goes. Once enabled globally, OSPF restart and helper modes are enabled by default. The GR options for OSPF are shown:

```
{master}[edit]
jnpr@R1-RE0# set protocols ospf graceful-restart ?
Possible completions:
+ apply-groups         Groups from which to inherit configuration data
+ apply-groups-except  Don't inherit configuration data from these groups
  disable              Disable OSPF graceful restart capability
> helper-disable       Disable graceful restart helper capability
  no-strict-lsa-checking  Do not abort graceful helper mode upon LSA changes
  notify-duration      Time to send all max-aged grace LSAs (1..3600 seconds)
  restart-duration     Time for all neighbors to become full (1..3600 seconds)
{master}[edit]
jnpr@R1-RE0# set protocols ospf graceful-restart
```

The `no-strict-lsa-checking` option is designed to work around the issue of a helping router aborting GR when it receives a hello from a restarting router that does not list the helper as a neighbor. By default, strict LSA checking is enabled. You should enable this option when helper routers don't support the newer graceful restart signaling approach, which is designed to solve this very issue. The `restart-duration` parameter specifies how long the restart event can last, and becomes the value that the restarting router places into its grace-LSA to begin a restart. The default is 180 seconds. The `notify-duration` specifies how long *after* the restart-duration has expired that the restarting router should announce the restart is complete by continuing to flush its grace-LSA. By default, this parameter is 30 seconds longer that the restart duration, or 220 seconds.

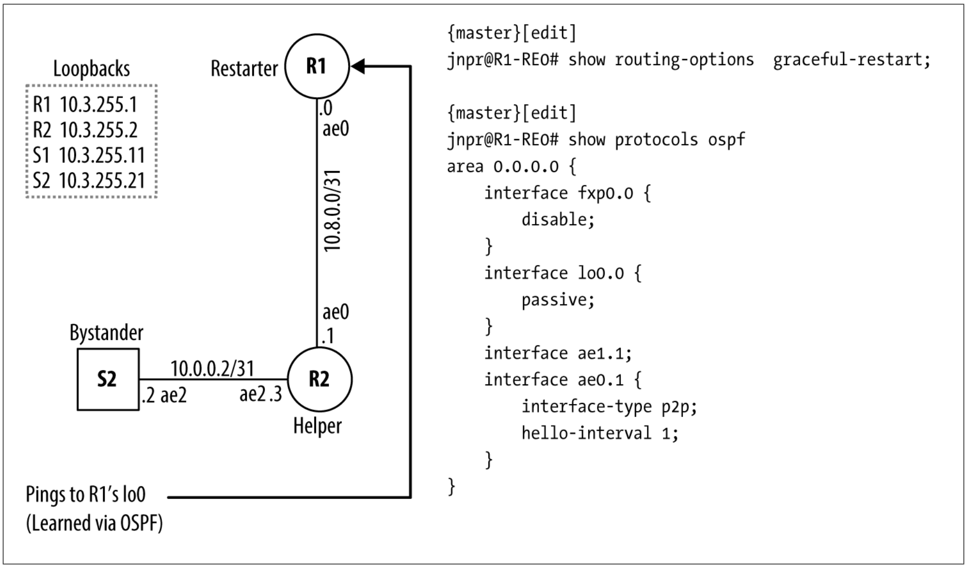
Use the `disable` statement to disable restarting functionality when GR is enabled globally. Note that helper functionality continues to work unless you specifically disable it with the `helper-disable` statement. Both standard (RFC 3623-based) and restart signaling-based helper modes are enabled by default, and you can optionally disable one, the other, or both helper modes:

```
{master}[edit]
jnpr@R1-RE0# show protocols ospf graceful-restart helper-disable ?
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
  both                 Disable helper mode for both the types of GR
```

```
restart-signaling    Disable helper mode for restart-signaling
standard             Disable helper-mode for rfc3623 based GR
|                    Pipe through a command
```

Verify OSPF GR

Operational verification of OSPF GR starts with the simplified OSPF network shown in Figure 9-5.



```
{master}[edit]
jnpr@R1-RE0# show routing-options  graceful-restart;

{master}[edit]
jnpr@R1-RE0# show protocols ospf
area 0.0.0.0 {
    interface fxp0.0 {
        disable;
    }
    interface lo0.0 {
        passive;
    }
    interface ae1.1;
    interface ae0.1 {
        interface-type p2p;
        hello-interval 1;
    }
}
```

Figure 9-5. OSPF GR Topology.

To best show the benefits of GR, we begin with graceful restart off on all routers; the result is a bunch of routers that can perform the helper function but none that can actually do a graceful restart. The configuration of R2 is shown:

```
{master}[edit]
jnpr@R2-RE0# show routing-options

{master}[edit]
jnpr@R2-RE0# show protocols ospf
traceoptions {
    file ospf_trace size 10m;
    flag lsa-update detail;
    flag graceful-restart detail;
    flag route detail;
}
area 0.0.0.0 {
    interface lo0.0 {
        passive;
    }
    interface ae0.1 {
        interface-type p2p;
        hello-interval 1;
    }
    interface ae2.0 {
        interface-type p2p;
        hello-interval 1;
    }
}
```

Of note is the absence of any global or OSPF-specific restart configuration. In other words, this is factory default from a GR perspective. In this example, the interfaces have been designated as type point-to-point, which dispenses with all that DR/BDR stuff, and a very short hello timer of 1 second is set to ensure rapid neighbor detection and adjacency formation when things are up and an equally rapid loss of adjacencies when things are down.

Keep in mind that BFD is not enabled in this scenario; normally, you would use BFD if such rapid failure detection is desired, but BFD is not compatible with GR, as described later in the section on NSR. These settings also work to make recovery faster due to rapid neighbor discovery and the ability to bypass the wait period to determine if a DR/BDR already exists, so this configuration cuts both ways, so to speak.

Note that tracing is enabled at R2 (and S2) to allow monitoring of key events related to LSA flooding, GR, and OSPF route changes.

We begin with confirmation of the expected OSPF adjacencies and LSDB contents at R2:

```
{master}[edit]
jnpr@R2-RE0# run show ospf neighbor
Address          Interface              State     ID               Pri  Dead
10.8.0.0         ae0.1                  Full      10.3.255.1       128     3
10.0.0.2         ae2.0                  Full      10.3.255.21      128     3

{master}[edit]
jnpr@R2-RE0# run show ospf database

    OSPF database, Area 0.0.0.0
 Type       ID               Adv Rtr           Seq      Age  Opt  Cksum  Len
Router   10.3.255.1       10.3.255.1       0x80000006   664  0x22 0xba6   60
Router  *10.3.255.2       10.3.255.2       0x800000a1   172  0x22 0xd5eb  84
Router   10.3.255.21      10.3.255.21      0x80000065  2879  0x22 0xa367  72
```

As expected, both adjacencies are up, and the single area OSPF network with all point-to-point interface types results in a single type 1 Router LSA for each OSPF node.

An Ungraceful Restart. The stage is now set to demonstrate the effects of a routing restart when GR is *not* enabled. You begin by verifying GR is disabled at R1:

```
{master}[edit]
jnpr@R1-RE0# run show route instance detail master
master:
  Router ID: 10.3.255.1
  Type: forwarding        State: Active
  Tables:
    inet.0                 : 21 routes (21 active, 0 holddown, 0 hidden)
```

The master instance does not display any restart duration, thus confirming GR is disabled globally. In addition, the OSPF overview at R1 does not list any GR information (but likely should, as helper mode is in effect):

```
{master}[edit]
jnpr@R1-RE0# run show ospf overview
Instance: master
  Router ID: 10.3.255.1
  Route table index: 0
  LSA refresh time: 50 minutes
  Area: 0.0.0.0
    Stub type: Not Stub
    Authentication Type: None
    Area border routers: 0, AS boundary routers: 0
    Neighbors
      Up (in full state): 1
  Topology: default (ID 0)
    Prefix export count: 0
    Full SPF runs: 4
    SPF delay: 0.200000 sec, SPF holddown: 5 sec, SPF rapid runs: 3
    Backup SPF: Not Needed
```

Pings are started at S2 to R1's lo0, as learned via OSPF:

```
{master:0}[edit]
jnpr@SW2-RE0# run show route 10.3.255.1

inet.0: 15 destinations, 15 routes (15 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

10.3.255.1/32      *[OSPF/10] 00:38:47, metric 2
> to 10.0.0.3 via ae2.0

jnpr@SW2-RE0# run ping 10.3.255.1 rapid count 2000
PING 10.3.255.1 (10.3.255.1): 56 data bytes
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!<ommitted for brievity>
```

With pings under way, the routing process on R1 is restarted. Note at this time you are also monitoring the OSPF trace log at R2, which is now empty given the network is still stable:

```
{master}[edit]
jnpr@R2-RE0#
*** 'ospf_trace' has been truncated - rewinding ***

*** monitor and syslog output enabled, press ESC-Q to disable ***
```

R1 has its routing restarted:

```
{master}[edit]
jnpr@R1-RE0# run restart routing
Routing protocols process started, pid 22832
```

```
{master}[edit]
jnpr@R1-RE0#
```

R2 is quick to notice the change and floods an updated router LSA:

```
jnpr@R2-RE0#
*** ospf_trace ***
Feb  6 13:33:00.561870 RPD_OSPF_NBRDOWN: OSPF neighbor 10.8.0.0 (realm ospf-v2 ae0.1
  area 0.0.0.0) state changed from Full to Down due to InActiveTimer
  (event reason: neighbor was inactive and declared dead)
Feb  6 13:33:00.562098 ospf_set_lsdb_state: Router LSA 10.3.255.2 adv-rtr 10.3.255.2
  state QUIET->GEN_PENDING
Feb  6 13:33:00.562107 OSPF trigger router LSA 0x93201d0 build for area 0.0.0.0
  lsa-id 10.3.255.2
Feb  6 13:33:00.562113 ospf_trigger_build_telink_lsas : No peer found
Feb  6 13:33:00.562185 OSPF restart siganling: Add LLS data for Hello packet on
  interface ae0.1.
Feb  6 13:33:00.612268 ospf_set_lsdb_state: Router LSA 10.3.255.2 adv-rtr 10.3.255.2
  state GEN_PENDING->QUIET
Feb  6 13:33:00.612295 OSPF built router LSA, area 0.0.0.0, link count 4
Feb  6 13:33:00.612367 OSPF sent LSUpdate 10.0.0.3 -> 224.0.0.5 (ae2.0 IFL 329
  area 0.0.0.0)
Feb  6 13:33:00.612375   Version 2, length 100, ID 10.3.255.2, area 0.0.0.0
Feb  6 13:33:00.612380   adv count 1
Feb  6 13:33:00.814328 CHANGE   10.3.255.1/32       nhid 579 gw 10.8.0.0
         OSPF     pref 10/0 metric 1/0 ae0.1 <Delete Int>
Feb  6 13:33:00.814365 rt_close: 1/1 route proto OSPF
Feb  6 13:33:00.814365
Feb  6 13:33:00.814416 rt_flash_update_callback: flash OSPF (inet.0) start
Feb  6 13:33:00.814422 Starting flash processing for topology default
Feb  6 13:33:00.814431 Finished flash processing for topology default
Feb  6 13:33:00.814438 rt_flash_update_callback: flash OSPF (inet.0) done
```

And the expected OSPF connectivity outage at S2 is confirmed:

```
!........ping: sendto: No route to host
.ping: sendto: No route to host
.ping: sendto: No route to host
.ping: sendto: No route to host
.ping: sendto: No route to host
.ping: sendto: No route to host
.ping: sendto: No route to host
.ping: sendto: No route to host
.ping: sendto: No route to host
.........ping: sendto: No route to host
.ping: sendto: No route to host
.ping: sendto: No route to host
.ping: sendto: No route to host
.ping: sendto: No route to host
.ping: sendto: No route to host
.ping: sendto: No route to host
.ping: sendto: No route to host
.ping: sendto: No route to host
.ping: sendto: No route to host
.!!<results ommitted for brievity>
```

It's pretty clear that with GR the loss of R1's adjacency to R2 was quickly noted, and the dataplane took the hit.

A Graceful Restart. The network is now altered to enable GR at R1. In most cases, you will want to enable GR on all routers, but in this case we know that only R1 is expected to restart anytime soon, so we rely on the default helper mode in the other routers, as enabled by default:

```
{master}[edit]
jnpr@R1-RE0# set routing-options graceful-restart

{master}[edit]
jnpr@R1-RE0# commit
re0:
. . .
```

Simple enough, right? You again confirm global and OSPF-level GR status:

```
{master}[edit]
jnpr@R1-RE0# run show route instance detail master
master:
  Router ID: 10.3.255.1
  Type: forwarding        State: Active
  Restart State: Pending  Path selection timeout: 300
  Tables:
    inet.0                 : 21 routes (21 active, 0 holddown, 0 hidden)
    Restart Complete

{master}[edit]
jnpr@R1-RE0# run show ospf overview
Instance: master
  Router ID: 10.3.255.1
  Route table index: 0
  LSA refresh time: 50 minutes
  Restart: Enabled
    Restart duration: 180 sec
    Restart grace period: 210 sec
    Graceful restart helper mode: Enabled
    Restart-signaling helper mode: Enabled
  Area: 0.0.0.0
    Stub type: Not Stub
    Authentication Type: None
    Area border routers: 0, AS boundary routers: 0
    Neighbors
      Up (in full state): 1
  Topology: default (ID 0)
    Prefix export count: 0
    Full SPF runs: 5
    SPF delay: 0.200000 sec, SPF holddown: 5 sec, SPF rapid runs: 3
    Backup SPF: Not Needed
```

The displays confirm that restart is now in effect and also show the main instance is pending the completion of its global restart timer. The master instance goes complete after initial GR activation 300 seconds later:

```
{master}[edit]
jnpr@R1-RE0# run show route instance detail master
master:
  Router ID: 10.3.255.1
  Type: forwarding        State: Active
  Restart State: Complete Path selection timeout: 300
  Tables:
    inet.0                 : 21 routes (21 active, 0 holddown, 0 hidden)
    Restart Complete
```

A quick traffic monitor on the ae0 interface at R1 confirms that restart signaling is in effect, as indicated by the presence of the LLS TLV. Note that R2 is sending the same options, confirming that GR helper mode is enabled there (by default):

```
13:42:26.028961  In IP (tos 0xc0, ttl   1, id 33165, offset 0, flags [none],
   proto: OSPF (89), length: 80) 10.8.0.1 > ospf-all.mcast.net: OSPFv2, Hello,
  length 60 [len 48]
        Router-ID 10.3.255.2, Backbone Area, Authentication Type: none (0)
        Options [External, LLS]
          Hello Timer 1s, Dead Timer 4s, Mask 255.255.255.254, Priority 128
          Neighbor List:
            10.3.255.1
          LLS: checksum: 0xfff6, length: 3
            Extended Options (1), length: 4
              Options: 0x00000001 [LSDB resync]
13:42:26.344602 Out IP (tos 0xc0, ttl   1, id 16159, offset 0, flags [none],
   proto: OSPF (89), length: 80) 10.8.0.0 > ospf-all.mcast.net: OSPFv2,
   Hello, length 60 [len 48]
        Router-ID 10.3.255.1, Backbone Area, Authentication Type: none (0)
        Options [External, LLS]
          Hello Timer 1s, Dead Timer 4s, Mask 255.255.255.254, Priority 128
          Neighbor List:
            10.3.255.2
          LLS: checksum: 0xfff6, length: 3
            Extended Options (1), length: 4
              Options: 0x00000001 [LSDB resync]
```

The graceful restart tracing in effect at R2 also confirms LLS-based GR, as the following is noted during adjacency formation:

```
Feb  6 15:09:19.250125 RPD_OSPF_NBRUP: OSPF neighbor 10.0.0.2 (realm ospf-v2 ae2.0
  area 0.0.0.0) state changed from Init to ExStart due to 2WayRcvd
  (event reason: neighbor detected this router)
Feb  6 15:09:19.250140 OSPF restart siganling: Send DBD with LR bit on to nbr
  ip=10.0.0.2 id=10.3.255.21.
Feb  6 15:09:19.250161 OSPF restart siganling: Add LLS data for Hello packet on
  interface ae2.0.
Feb  6 15:09:19.250193 OSPF restart siganling: Add LLS data for Hello packet on
  interface ae2.0.
```

With GR now enabled and confirmed, we once again perform a restart routing at R1. As before, the OSPF trace file is monitored at R2 and S2 is generating traffic to a destination on R1 that is learned through OSPF.

```
{master}
jnpr@R1-RE0>restart routing immediately
```

```
{master}
jnpr@R1-RE0>
```

And at R2 trace activities confirms the LLS-based GR event:

```
{master}[edit]
jnpr@R2-RE0#
*** 'ospf_trace' has been truncated - rewinding ***

*** ospf_trace ***
Feb 6 15:11:32 R2-RE0 clear-log[8549]: logfile cleared
Feb  6 15:12:13.042243 OSPF rcvd LSUpdate 10.8.0.0 -> 224.0.0.5 (ae0.1 IFL
  326 area 0.0.0.0)
Feb  6 15:12:13.042446   Version 2, length 64, ID 10.3.255.1, area 0.0.0.0
Feb  6 15:12:13.042451   checksum 0x0, authtype 0
Feb  6 15:12:13.042456   adv count 1
Feb  6 15:12:13.042488 OSPF LSA OpaqLoc 3.0.0.0 10.3.255.1 from 10.8.0.0 newer
  than db
Feb  6 15:12:13.042510 ospf_set_lsdb_state: OpaqLoc LSA 3.0.0.0 adv-rtr
  10.3.255.1 state QUIET->QUIET
Feb  6 15:12:13.042518 OSPF Restart: starting helper mode for neighbor 10.3.255.1
  on intf ae0.1 area 0.0.0.0
Feb  6 15:12:13.042524 OSPF Restart: grace timer updated to expire after
  208 seconds
Feb  6 15:12:14.042899 OSPF rcvd LSUpdate 10.8.0.0 -> 224.0.0.5 (ae0.1
  IFL 326 area 0.0.0.0)
Feb  6 15:12:14.043079   Version 2, length 64, ID 10.3.255.1, area 0.0.0.0
Feb  6 15:12:14.043084   checksum 0x0, authtype 0
Feb  6 15:12:14.043088   adv count 1
Feb  6 15:12:14.043113   Same as db copy
Feb  6 15:12:15.043627 OSPF rcvd LSUpdate 10.8.0.0 -> 224.0.0.5 (ae0.1 IFL
  326 area 0.0.0.0)
Feb  6 15:12:15.043818   Version 2, length 64, ID 10.3.255.1, area 0.0.0.0
Feb  6 15:12:15.043823   checksum 0x0, authtype 0
Feb  6 15:12:15.043827   adv count 1
Feb  6 15:12:16.045085 OSPF rcvd LSUpdate 10.8.0.0 -> 224.0.0.5 (ae0.1 IFL
  326 area 0.0.0.0)
Feb  6 15:12:16.045259   Version 2, length 64, ID 10.3.255.1, area 0.0.0.0
Feb  6 15:12:16.045264   checksum 0x0, authtype 0
Feb  6 15:12:16.045268   adv count 1
Feb  6 15:12:17.043150 OSPF rcvd LSUpdate 10.8.0.0 -> 224.0.0.5 (ae0.1 IFL
  326 area 0.0.0.0)
Feb  6 15:12:17.043348   Version 2, length 64, ID 10.3.255.1, area 0.0.0.0
Feb  6 15:12:17.043353   checksum 0x0, authtype 0
Feb  6 15:12:17.043357   adv count 1
Feb  6 15:12:18.042974 OSPF rcvd LSUpdate 10.8.0.0 -> 224.0.0.5 (ae0.1 IFL
  326 area 0.0.0.0)
Feb  6 15:12:18.043182   Version 2, length 64, ID 10.3.255.1, area 0.0.0.0
Feb  6 15:12:18.043187   checksum 0x0, authtype 0
Feb  6 15:12:18.043191   adv count 1
Feb  6 15:12:19.045018 OSPF restart siganling: Add LLS data for Hello packet
  on interface ae0.1.
Feb  6 15:12:19.046180 OSPF restart siganling: Send DBD with LR bit on to nbr
  ip=10.8.0.0 id=10.3.255.1.
Feb  6 15:12:19.046201 OSPF restart siganling: Add LLS data for DbD packet on
```

```
  interface ae0.1.
Feb  6 15:12:19.046285 OSPF restart siganling: Received DBD with LLS data from
  nbr ip=10.8.0.0 id=10.3.255.1.
Feb  6 15:12:19.088271 OSPF restart siganling: Received DBD with LLS data from
  nbr ip=10.8.0.0 id=10.3.255.1.
Feb  6 15:12:19.088309 OSPF restart siganling: Send DBD with LR bit on to nbr
  ip=10.8.0.0 id=10.3.255.1.
Feb  6 15:12:19.088331 OSPF restart siganling: Add LLS data for DbD packet on
  interface ae0.1.
Feb  6 15:12:19.127630 OSPF restart siganling: Received DBD with LLS data from
  nbr ip=10.8.0.0 id=10.3.255.1.
Feb  6 15:12:19.127726 OSPF sent LSUpdate 10.8.0.1 -> 224.0.0.5 (ae0.1 IFL
  326 area 0.0.0.0)
Feb  6 15:12:19.127735   Version 2, length 244, ID 10.3.255.2, area 0.0.0.0
Feb  6 15:12:19.127740   adv count 3
Feb  6 15:12:31.131050 OSPF rcvd LSUpdate 10.8.0.0 -> 224.0.0.5 (ae0.1 IFL
  326 area 0.0.0.0)
Feb  6 15:12:31.131249   Version 2, length 124, ID 10.3.255.1, area 0.0.0.0
Feb  6 15:12:31.131254   checksum 0x0, authtype 0
Feb  6 15:12:31.131258   adv count 2
Feb  6 15:12:31.131322 OSPF LSA OpaqLoc 3.0.0.0 10.3.255.1 from 10.8.0.0 newer
  than db
Feb  6 15:12:31.131336 ospf_set_lsdb_state: OpaqLoc LSA 3.0.0.0 adv-rtr 10.3.255.1
  state QUIET->QUIET
Feb  6 15:12:31.131343 OSPF Restart: exiting helper mode on Grace LSA purge for nbr
  10.3.255.1 (intf ae0.1 area 0.0.0.0)
Feb  6 15:12:31.131359 ospf_set_lsdb_state: OpaqLoc LSA 3.0.0.0 adv-rtr 10.3.255.1
  state QUIET->PURGE_PENDING
Feb  6 15:12:31.131391 OSPF LSA Router 10.3.255.1 10.3.255.1 from 10.8.0.0 newer
  than db
Feb  6 15:12:31.131407 ospf_set_lsdb_state: Router LSA 10.3.255.1 adv-rtr
  10.3.255.1 state QUIET->QUIET
Feb  6 15:12:33.132916 OSPF sent LSUpdate 10.0.0.3 -> 224.0.0.5 (ae2.0 IFL
  329 area 0.0.0.0)
Feb  6 15:12:33.133108   Version 2, length 88, ID 10.3.255.2, area 0.0.0.0
Feb  6 15:12:33.133113   adv count 1
Feb  6 15:12:41.132244 ospf_set_lsdb_state: OpaqLoc LSA 3.0.0.0 adv-rtr 10.3.255.1
  state PURGE_PENDING->QUIET
```

The "restart signaling" typo in the previous was already reported in PR-577297 and corrected in the v12.1 Junos release.

Given the GR was successful, it's a case of "look to see what didn't happen" over at S2, where no traffic is lost despite the restart of routing at R1. Given this, you can see where GR is often referred to as Nonstop Forwarding (NSF).

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!^C!
--- 10.3.255.1 ping statistics ---
36571 packets transmitted, 36571 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.609/1.601/76.257/2.700 ms
```

Graceful Restart Summary

GR, or NSF, helps improve availability by allowing uninterrupted dataplane forwarding through a control plane restart event. The restart helper routers assist the restarting routing in rebuilding its pre-restart RIB, so it can resume normal operation as soon as possible after a restart.

GR has the drawbacks of requiring protocol extensions, peer-router support in the form of helpers, and abortion if the helpers detect network instability during the restart. In addition, GR does not work well with BFD, so it's not a panacea all things HA. GR does have the advantage of not requiring dual REs/GRES, but is also capable of working with GRES for tolerance of RE failure as well as control plane resets.

Most networks today skip GR so they can instead deploy the next stage in routing HA known as Nonstop Routing (NSR). The next section picks up where GR leaves off with an in-depth exploration of NSR and NSB operation on MX routers.

Nonstop Routing and Bridging

NSR and NSB represent the current state of the HA art for Junos. The general concepts of NSR and NSB are very similar, so unless calling out specifics, the terms NSR and NSB are used interchangeably. While any dual RE router can avail itself of NSR, currently only MX platforms support NSB.

Unlike GR, which makes no bones about announcing a control plane restart, with the full expectation of gaining help from its neighbors, NSR is a completely self-contained solution. A successful NSR event has no externally visible symptoms. NSR does not require any protocol extensions, and there is no need for the helper role associated with GR; while the attached neighbors may well be GR capable and therefore able to provide a helper role, the nature of NSR's self-contained solution means that restart helper services are simply never needed.

In addition, a successful NSR event is not predicated on network stability during a switchover, a fact that greatly improves the chances of hitless switchover, when compared to GR.

NSR is the foundation upon which the In-Service Software Upgrade (ISSU) feature is built. If you plan on using ISSU, you need to have NSR configured also.

On routers that have logical systems configured on them, only the master logical system supports nonstop active routing.

Replication, the Magic That Keeps Protocols Running

The heart and soul of NSR in Junos is protocol replication. The basic concept is to replicate the actual protocol messages between the master and BU RE, including TCP connection state for BGP, so that at any given time both REs have the same protocol view. This requires that the BU RE run `rpd`, the routing daemon, on the replicated protocol messages to independently maintain a shadow copy of the RIB. The part about independence in the last sentence is a key point: it's not just a mirror of the master's state, which has the potential to mirror any bugs or defects that may have occurred due to corrupted memory/hardware, or perhaps just bad timing. Instead it's the actual protocol messages themselves that are replicated with the BU RE running its own routing daemon, and therefore its own SPF and BGP route selection algorithms, effectively eliminating the fate sharing that would exist if the state was simply mirrored.

Given the replicated protocol messages and common routing behavior on both REs, one generally expects that with NSR the active route selection will match between master and BU, but this is not always so. In some cases, a different LB hash decision may be made or a different equal cost route might be preferred based on message timing or whatever tie-breaking criteria are used. This can lead to a forwarding change at switchover, which is not to say there will be packet loss, so much as use of a different forwarding next-hop. To help avoid this behavior with BGP, which by default can use the relative time that a route was learned as a tie breaker, Juniper recommends that you add the `path-selection external-router-ID` statement at the `[edit protocolsbgp]` hierarchy to ensure consistent BGP path selection between the master and backup REs.

When all goes to plan in the event of an NSR-based GRES, the new master literally picks up where the old one left off, for example, by generating a TCP ACK for a BGP update that had just been received right as the old master failed. From the viewpoint of direct and remote protocol neighbors, nothing happens, thus the switchover is transparent and therefore a nonevent; technically speaking, and depending on scale, some peers may note a delay, or possibly miss a hello or protocol keepalive, but such events are normal in protocol operation and do not in themselves force tear down of a routing adjacency.

To help hide all signs of a switchover, Junos now defaults to distributed processing mode for most periodic protocol hello functions, whereby the hellos are generated within the PFE itself via the `ppmd` process. Running in distributed mode is especially critical for protocols like BFD or LACP, which tend to have rapid failure detection times and therefore don't take kindly to delayed hellos during an NSR or GRES event. With hellos autonomously generated by the PFE during the RE switchover window, when the RE itself cannot generate such packets, it lends itself to making NSR truly undetectable for the router's neighbors as (distributed) hellos are not even delayed, let alone missed.

The delegate-processing statement at the [edit routing-options ppm] hierarchy, which was used to enable distributed ppmd in Junos OS release v9.3 and earlier, has been deprecated. The ppmd process is distributed to the PFE by default in Junos OS release v9.4 and later. However, if you want the PPM process to run on the RE instead of the Packet Forwarding Engine, you can do so by including the no-delegate-processing statement at the [edit routing-options ppm] hierarchy level. Note that LACP can be forced to run in a centralized mode even though ppmd is set to distributed with a set protocols lacp ppm centralized, which if set will lead to LACP flap at GRES.

When an MX is part of a virtual chassis, you should configure a longer PPM distribution timer using a set routing-options ppm redistribution-timer 120 statement to ensure proper PPM processing during a VC GRES event. Refer to Chapter 6 for details. Note that the redistribution-timer statement is hidden in the 11.4 release.

Figure 9-6 begins the overview of BGP replication, which begins with TCP connection establishment.

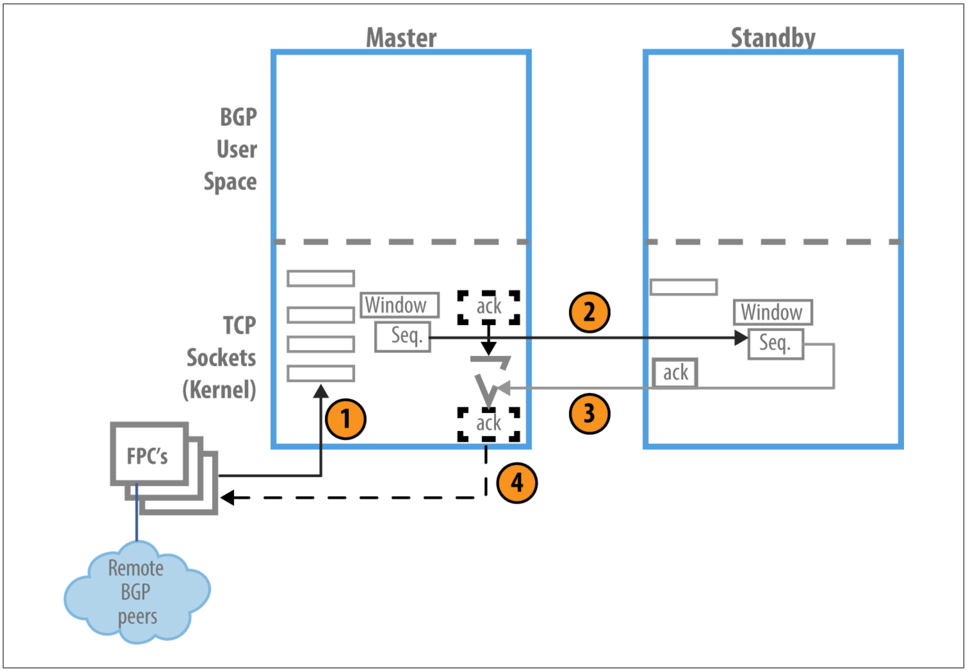


Figure 9-6. BGP Replication Part 1: TCP Socket State.

It's useful to review what actually happens in every step shown in the replication diagram:

- Step 1 in the figure represents an incoming TCP connection request (SYN Flag) to the BGP port (179).
- Step 2 shows the master kernel preparing the resulting SYN + ACK segment (shown simply as "ACK" in the figure), but does not actually place the ACK onto the wire because at this stage the backup RE kernel has not yet acknowledged the new socket state. At the same time, a kernel socket replication function is used to convey the SYN segment to the kernel on the backup RE, where matching state can now be established.
- Step 3 shows the backup RE kernel ACK message sent back to the master RE kernel.
- Step 4 shows the master kernel can, once it has received the ACK message from backup RE, transmit the ACK segment back to the connection's originator, thus completing step 2 of the TCP three-way handshake for connection establishment.

Waiting for the BU RE to catch up before moving forward is a key aspect of Juniper's NSR solution.

By ensuring that any traffic to be transmitted is replicated before it's actually placed on the wire, and that all received traffic is replicated before it's acknowledged by the master TCP socket, ensures that in all cases of failover, the secondary application is as informed about the state of the network as any remote peer it ends up communicating with.

Once the TCP connection is established, BGP traffic can be sent and received by the routing process on the master RE through its TCP socket. Meanwhile, the BU routing process eavesdrops on these BGP messages through its read-only replicated sockets, as shown in Figure 9-7.

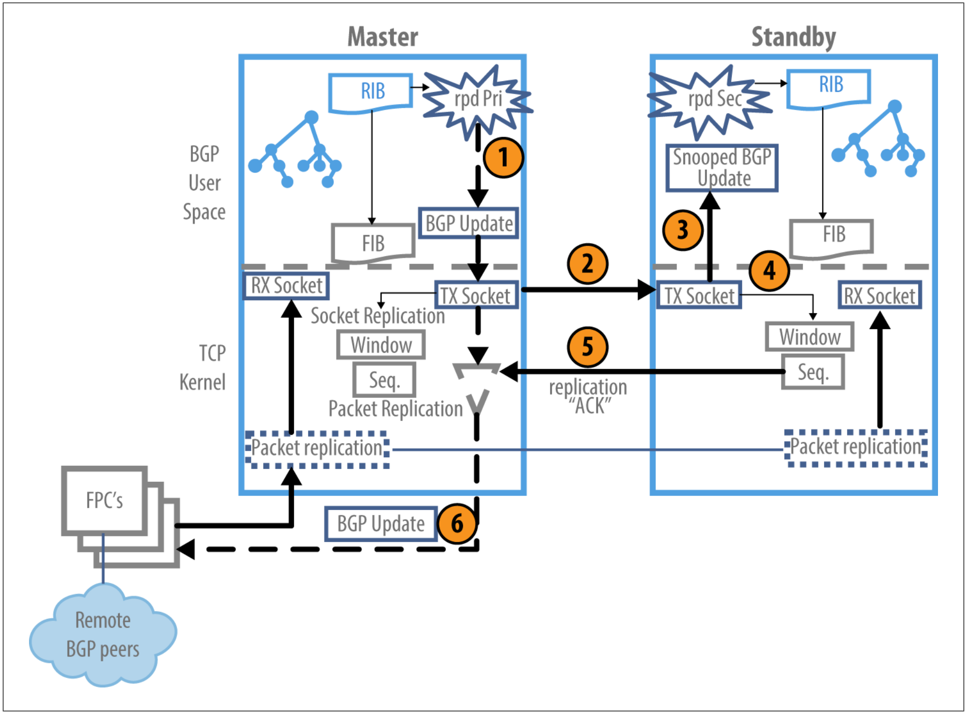


Figure 9-7. BGP Replication Part 2: Transmit Side Snooping and Replication.

The figure focuses on how a BGP update message is ultimately transmitted to a remote peer. Again, it's useful to walk every step through the process to fully understand what happens behind the curtains:

- Step 1 (as they are wont to do): Where a RIB update results in the master routing process creating a BGP update message, perhaps to withdraw a route that has become unreachable, or maybe to update the route's attributes due to a policy change that indicates a community should now be attached.
- Step 2: The BGP update enters the socket replication layer, which redirects the update to the read-only copy of the transmit socket maintained on the backup RE.
- Step 3: The secondary RPD process snoops the transmitted BGP messages, so it can keep in lock-step with the master by performing any updates on its internal BGP data structures or to its copies of the RIB/FIB. For example, it might update its BGP RIB-OUT to indicate that a new community is attached to the corresponding NLRI for the related peer. As a result, the operator can expect to see the same results in the output of a `show route advertising-protocol bgp <peer>` command for that prefix on both the master and backup REs. This is critical, as the backup could find itself active at any given time and it must have identical copies of both

the BGP-IN and BGP-OUT RIBs to ensure the mastership change remains undetected.

- Step 4: The socket replication layer updates the TCP transmit socket state on both REs.
- Step 5: The primary kernel waits for the replication acknowledgment; once received, the master kernel will be able to send the actual BGP update to the remote peer.
- Step 6: After successful reception of the replication ack as per step 5, the master kernel proceeds to send the actual BGP update to the remote peer.

This process ensures that when a switchover occurs, the new primary is guaranteed to have as much information as the remote peer, again a necessary state to ensure a successful hitless switch in mastership. While the details are beyond our scope, it's noted that the replication process supports flow control and that each replicated protocol uses its own sockets and native messages as part of its replication process. This allows each protocol to replicate at its own pace, and ensures no messages are lost during periods of heavy CPU usage and/or when a large volume of protocol updates are occurring, such as when a network is undergoing reconvergence.

While not detailed, a similar process occurs for received BGP updates, including a packet replication function, which is independent of the socket replication layer's job of keeping the TCP layer's connection parameters synchronized. The replication process for transmitted BGP traffic ensures that the snooping process is synchronized before updates are transmitted to remote peers. In a similar fashion, the receive replication process ensures that the BU routing and replication functions have acknowledged received updates before the related TCP acknowledgement is sent by the master RE's kernel. With this multiple synchronization point mechanism, any packets or replication traffic that are lost due to being "in-flight" as the switchover occurs are naturally recovered by the built-in protocol mechanisms. For example, if the actual BGP update shown at step 6 happens to be corrupted at the moment of switchover, the effect is no different than any other lost TCP segment (or acknowledgement). In this case, the new master has already seen and acknowledged both the traffic and resulting sending socket state via replication before the failover. Therefore, its TCP socket is running the same ACK timer as was in effect on the previous master. Should the timer expire, the new master retransmits any unacknowledged traffic, which this time makes it to the remote peer, who returns the expected TCP ACK and all is well. To the remote peers, this is simply another case of best-effort IP dropping traffic, something that TCP is well-equipped to deal with.

Figure 9-8 shows the state of affairs after a failure in the primary RE has resulted in an NSR-based GRES.

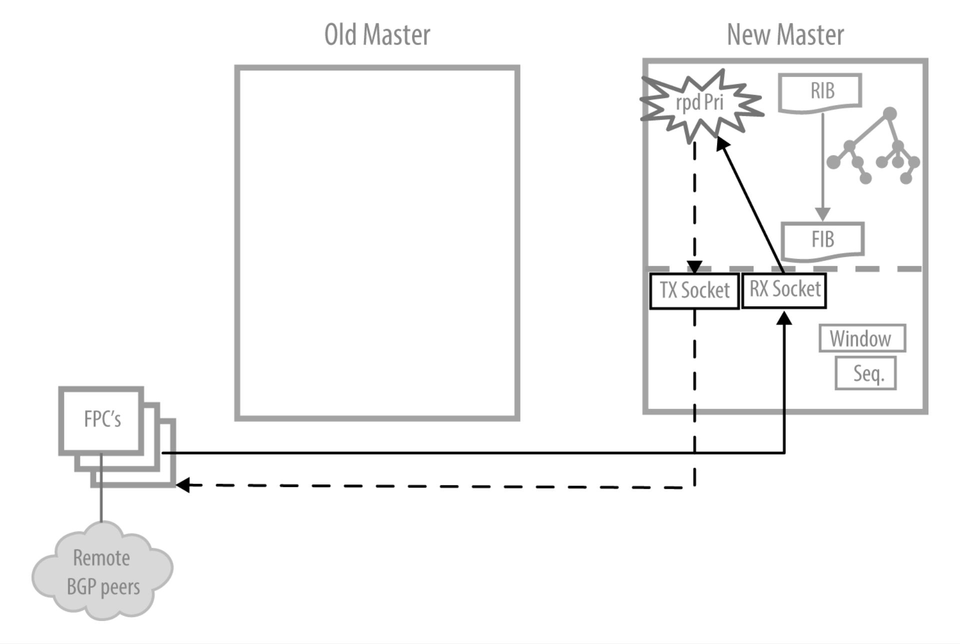


Figure 9-8. BGP Replication Part 3: Meet the New Boss.

The figure shows the former BU RE now functioning as master. Its previously read-only RPD sockets are now able to be written to, and the new master's kernel is allowed to send traffic onto the wire, allowing it to pick up where the last master left off. When all goes to plan, no protocol sessions are reset. In fact, remote peers never even notice they are now communicating with a different RE. With no control plane perturbance, there is no need to alter the dataplane, so packet forwarding continues as before the switchover, which in most cases means 0 packet loss through the switchover event as well.

> Though not strictly required, running the same Junos version on both REs when NSR is enabled is recommended and helps to reduce the chances of a version-based replication incompatibility that could cause NSR to fail during a GRES.

Nonstop Bridging

Juniper's NSR and GRES features have proven successful in the field. As the company expanded into the Enterprise and Layer 2 switching markets, it was only a matter of time before customers would demand similar HA features for switched networks. Enter Nonstop Bridging (NSB), a feature currently only supported on the MX family of routers and EX switches. Figure 9-9 details key concepts of the Junos NSB implementation.

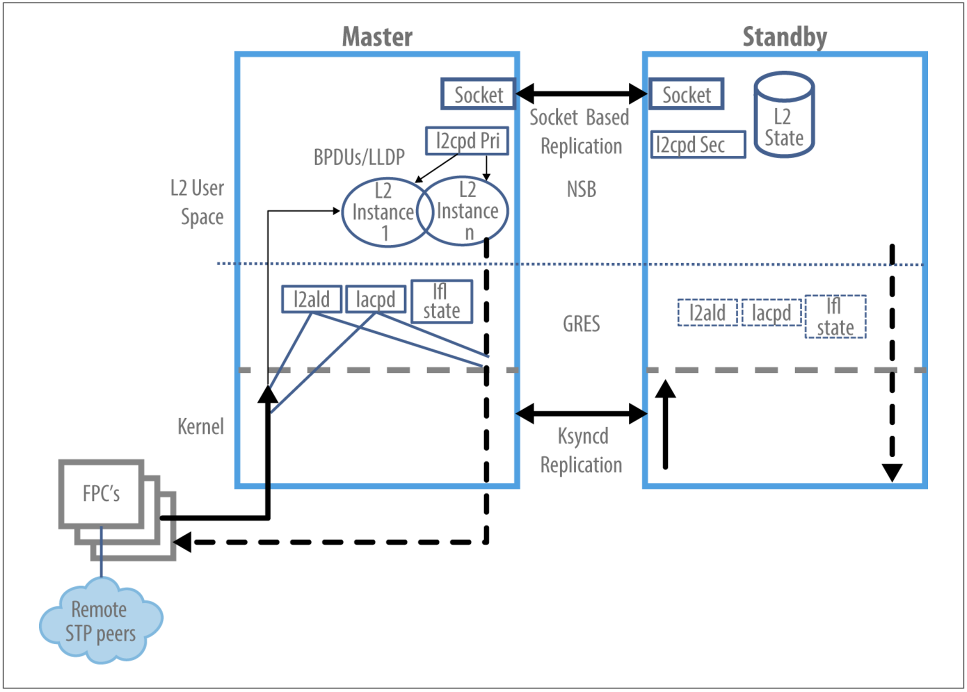


Figure 9-9. Nonstop Bridging.

When you enable NSB, the Layer 2 Control Protocol Daemon (`l2cpd`) runs on both the primary and BU REs, much as was the case with the routing daemon (`rpd`) for NSR. The `l2cpd` process implements the various forms of Spanning Tree Protocol (STP), making it the "Layer 2 bridging module" for Junos; this daemon also provides the LLDP service on MX routers.

In NSB mode, `l2cpd` is started on the backup RE. Once running, it establishes a socket connection to the l2cpd process running on master RE to synchronize xSTP protocol state. For STP and RSTP, this state includes the root bridge ID, the root path cost, and the STP state for each IFD/IFL in each L2 control instance (main or virtual-bridge). For MSTP, the CIST root identifier, the CIST internal root path cost, the MSTI regional root, the MSTI internal root path cost, the MSTI remaining hops, and the STP state for each MSTI are mirrored. This information, when combined with other state, such as interface information mirrored in the kernel via GRES, is enough for the Layer 2 control process to begin forwarding in the same state as the former master immediately after a switchover.

NSB Only Replicates Layer 2 State

Unlike the Layer 3 NSR implementation, where the same Layer 3 protocol state is expected to be found on both REs, on the MX the Layer 2 protocol state machine is not run in the backup RE. Instead, only the states are synchronized. This state primes the initial starting condition for the various xSTP processes once a switchover occurs. Just as with NSR, while the xSTP processes are being kickstarted on the new master, the prior forwarding state in the PFE is preserved, meaning there should be no dataplane hit for Layer 2 traffic. In contrast, when only GRES is configured, all xSTP ports are initially placed into blocking as the `l2cpd` process starts on the new master post switchover. This is the only option if loops are to be avoided, given that without NSB the new master cannot benefit from knowing the previous master's Layer 2 state.

NSB and Other Layer 2 Functions

The figure shows how some other Layer 2 protocols functions, such as Ethernet OAM, LACP, and the Layer 2 address learning daemon (`l2ald`), have their state replicated as part of GRES infrastructure. The address learning daemon is used for bridge instance and VPLS MAC address-related learning functions. Note that, even when NSR is configured, `l2ald` runs only on the master RE. In contrast, the `lacpd` daemon runs on both REs even when GRES/NSR is not configured, but hitless switchover when AE bundles are present still requires an operational GRES infrastructure. It's normal to get no output from a `show lacp interfaces` command on the BU RE even when NSR and GRES are in effect. LACP replication being handled as part of GRES is just one of those things you have to take on faith with Junos.

When you combine GRES with NSB, you can expect hitless switchovers for LAG/LACP bundles, MAC learning, and xSTP state. After a switchover, LLDP state is rebuilt, but this does not cause any dataplane impact.

Current NSR/NSB Support

Junos currently offers NSR support for a wide range of protocols, and as a strategic feature, NSR support is expected to continually evolve with a active development of new features and improved functionality. As of the 11.4 release, NSR is broadly supported for:

> Aggregate and static routes, RIP, RIPng, OSPF, OSPF3, IS-IS, MP-BGP (multiple families)
> RSVP on ingress, transit (and egress) provider edge (PE) routers, p2mp
> LDP on ingress, transit, and egress PE routers
> Layer 2 VPN, Layer 3 VPNs, 6VPE (Inet6 tunneling over MPLS)
> PIM sparse and dense mode (main instance), IPv4/IPv6
> IGMP/MLD
> BFD for a number of different client protocols, to includes static route, OSPF, OSFP3, BGP, IS-IS, RSVP, etc.
> VPLS routing instances
> VRRP (IPv4/IPv6)
> LACP/LAG

Bridging (xSTP)
LLDP
Ethernet Operation, Administration, and Management (OAM) as defined 8.5 by IEEE 802.3ah and IEEE 802.1ag 9.0

Table 9-2 lists key NSR feature support as a function of major Junos release.

Table 9-2. NSR Feature Support by Release.

Feature	Release
Aggregated Ethernet interfaces with Link Aggregation Control Protocol (LACP)	9.4 or later
Bidirectional forwarding detection (BFD) for BGP, IS-IS, OSPF/OSPFv3, or PIM	8.5 or later (OSPF3 sessions are RE based)
BGP	8.4 or later
IS-IS	8.4 or later
LDP	8.4 or later
LDP-based virtual private LAN service (VPLS)	9.3 or later
LDP OAM (operation, administration, and management) features	9.6 or later
Layer 2 circuits	on LDP-based VPLS, 9.2 or later
	on RSVP-TE LSP, 11.1 or later
Layer 2 VPNs	9.1 or later
Layer 3 VPNs *	9.2 or later (as of 11.4, Layer 3 VPN support does not include dynamic GRE tunnels, multicast VPNs [NGEN MVPN], or BGP flow routes)
OSPF/OSPFv3	8.4 or later
Protocol Independent Multicast (PIM)	(for IPv4) 9.3 or later
	(for IPv6) 10.4 or later
RIP and RIP next generation (RIPng)	9.0 or later
RSVP-TE LSP	9.5 or later
VPLS	VPLS (LDP-based) 9.1 or later
	(RSVP-TE-based) 11.2 or later
NSB (xSTP)	11.4

BFD and NSR/GRES Support

Nonstop active routing supports the bidirectional forwarding detection (BFD) protocol, which uses the topology discovered by routing protocols to confirm forwarding state between protocol neighbors, with rapid failure detection and resulting protocol session teardown in the event of faults. Generally speaking, the BFD protocol can be run from the RE itself, or from within the PFE using distributed mode where its hellos are handled by the ppmd process. Distributed mode offers the advantages of increased BFD session counts that can in turn support reduced (shorter) detection timers and is the default mode of BFD processing. When a BFD session is distributed to the Packet Forwarding Engine (the default), BFD packets continue to be sent during a GRES event. With NSR enabled and BFD in distributed mode, the BFD session states are not reset during a GRES event.

However, not all BFD sessions can be distributed. If a nondistributed BFD session is to be maintained through a switchover, you must ensure that failure detection time is greater than the RE switchover time given that no BFD packets can be sent until the new master has finished coming online. The following BFD sessions cannot be distributed to the Packet Forwarding Engine in release 11.4, so make sure you set appropriately long detection times to ensure they can survive a GRES:

- All multihop sessions (i.e., BFD protecting loopback-based BGP sessions)
- All tunnel-encapsulated sessions
- Sessions that run over Integrated Routing and Bridging (IRB) interfaces
- IPV6 sessions that use link local addresses (i.e., OSPF3)

BFD Scaling with NSR. BFD can be configured to run in an aggressive manner for really fast error detection, but as with all things the compromise here is that faster sessions require commensurately higher system resources for their support. Specifying a minimum interval for BFD less than 100 ms for RE-based sessions and 10 ms for distributed BFD sessions can cause undesired BFD flapping even when no switchovers are taking place. Depending on your network environment, you may have to increase BFD detection timers to ensure stability even during periods of heavy churn, such as when recovering from a network failure. Carefully consider these points when selecting the BFD minimum interval:

- For large-scale network deployments with up to 300 BFD sessions per FPC, specify a minimum interval of 100 ms for distributed BFD sessions and 300 ms for RE-based sessions; note that per FPC counts don't apply for RE based sessions. A setting of 900 ms is suggested to support up to 900 BFD sessions per FPC (distributed mode). These numbers are based on a multiplier of three.
- For RE-based BFD sessions to remain up during a GRES/NSR event, you must specify a minimum interval of 2500 ms and a multiplier of at least three. Juniper supports up to 7,500 RE-based sessions at the 2500 ms setting through a NSR. Distributed sessions are not impacted by NSR and should have their minimum intervals based on session scale, as noted previously.

BFD and GR—They Don't Play Well Together. Before moving on, it bears noting again that BFD and GR are considered mutually exclusive. Juniper's recommendation is that you not run both at the same time; this was mentioned in the previous section on GR, but, having reached this stage an explanation for this nonintuitive restriction can be offered. The issue is somewhat akin to trying to run GR and NSR at the same time, which is not possible as one seeks to announce a control plane fault while the other endeavors hide it. The same is true for BFD and GR. Here, GR is attempting to preserve the dataplane while admitting to a control plane fault, whereas BFD exists to rapidly spot any issue in the dataplane and then direct traffic *away* from the problem area by bringing down the affected protocol sessions.

The basic problem when you combine GR and BFD is best understood with a specific example of a BFD-protected IS-IS adjacency. Here, we have somewhat of a bootstrap issue, where BFD learns its end points from the IS-IS adjacency itself. This is all well and fine, until that adjacency is lost as part of a control plane restart, which is expected with GR. The obvious problem is that the (expected) loss of IS-IS state in the restart triggers the BFD adjacency to drop, which when detected by the remote peer forces a topology change with resulting IS-IS LSP flooding, that in turns causes GR to abort. So go the best laid plans, of mice and men . . . Note that with a NSR event the IS-IS adjacency is not lost, and the `ppmd` process distributed in the PFE keeps hellos flowing such that no flap at either the IS-IS or BFD layers is expected, making the switchover transparent.

NSR and BGP

While BGP has been NSR supported for a long time, it's one of the primary workhorses behind Junos, and as such is often upgraded with new features or capabilities. For example, the `inet-mdt` family that was added in release v9.4 for Rosen7 MVPN support or the newer `inet-mvpn` family added for next-generation MVPNs both lacked NSR support in their initial releases.

When you configure a BGP peer with a nonsupported address family, you can expect the corresponding session to be idled on the BU RE. Upon switchover, the idled BGP sessions have to be reestablished, and the result is a significant data plane hit for *all* NLRI associated with the affected peering sessions until the network reconverges. As an example, here the IBGP session at R1 is updated to include the NGEN MVPN related `inet-mvpn` family:

```
jnpr@R1-RE1# show protocols bgp group int
type internal;
local-address 10.3.255.1;
family inet {
    unicast;
}
family inet-mvpn {
    signaling;
}
bfd-liveness-detection {
    minimum-interval 150;
    multiplier 3;
}
neighbor 10.3.255.2;
```

The BGP session is up on the master RE:

```
{master}[edit]
jnpr@R1-RE1# run show bgp summary
Groups: 2 Peers: 2 Down peers: 1
Table          Tot Paths  Act Paths Suppressed    History Damp State    Pending
inet.0               100        100          0          0          0          0
bgp.mvpn.0             0          0          0          0          0          0
Peer                     AS      InPkt     OutPkt    OutQ   Flaps Last Up/Dwn State|
  #Active/Received/Accepted/Damped...
10.3.255.2      65000.65000         12         12       0       1        4:15 Establ
  inet.0: 0/0/0/0
  bgp.mvpn.0: 0/0/0/0
```

But as predicted, the session is idle on the BU RE, confirming that a non-NSR-supported protocol family is in effect. This state provides an invaluable clue to the operator that this BGP session will flap at switchover, affecting all associated NLRI from all families used on the peering:

```
{backup}[edit]
jnpr@R1-RE0# run show bgp summary
Groups: 2 Peers: 2 Down peers: 2
Table          Tot Paths  Act Paths Suppressed    History Damp State    Pending
inet.0                 0          0          0          0          0          0
bgp.mvpn.0             0          0          0          0          0          0
Peer                     AS      InPkt     OutPkt    OutQ   Flaps Last Up/Dwn State|
  #Active/Received/Accepted/Damped...
10.3.255.2      65000.65000          3          2       0       2        4:44 Idle
```

If that is not bad enough, consider that even when a family is NSR supported, that does not in itself mean the related service is expected to be hitless. For example, consider again the case of a rosen7-based MVPN and the related `inet-mdt` family. Initially in v9.4, the use of this family would idle the session on the BU RE, as shown previously for the `inet-mvpn` family in the 11.4R1.9 release. Sometime around v10.4, “NSR support” was added to the `inet-mdt` family, which meant that the BU RE no longer shows the session as idle. However, the `inet-mdt` family is used to support MVPN (PIM in a routing-instance), and as of 11.4 PIM in instances is *not* an NSR-supported feature.

So, what does all this mean? If you use 11.4 and have Rosen7 MVPN and NSR configured, you can expect all BGP sessions to be replicated on the BU RE, and therefore no session flap at switchover. Services like L3VPN unicast will be hitless, whereas the MVPN service experiences an outage due to the loss of native multicast state (joins and learned RP information) during the switchover. Current PIM NSR support is covered in detail in a following section.

As of the 11.4 release, the following BGP address families are NSR supported:

inet unicast
inet labeled-unicast
inet multicast
inet6 labeled-unicast
inet6 unicast
route-target

l2vpn signaling
inet6-vpn unicast
inet-vpn unicast
inet-mdt
iso-vpn

> Note that a draft-Rosen Multicast VPN (MVPN) configuration fails to commit when nonstop active routing for PIM is enabled in the 11.4 release. You must disable nonstop active routing for PIM if you need to configure a draft-Rosen MVPN.

> Address families are supported only on the main instance of BGP; only unicast is supported on VRF instances.

The list is long and includes most all the popular BGP families. The most noticeable exceptions for support with NSR in the 11.4 release are `inet flow`, as used to support the BGP flowspec feature, and the `inet-mvpn` and `inet6-mvpn` families, used for (BGP-based) NGEN MVPNs.

Note that BGP route dampening does not work on the backup RE when NSR is enabled. After an NSR event, damping calculations are rerun based on current flap state.

NSR and PIM

PIM is a complex protocol with many options. Nonstop active routing support varies for different PIM features. The features fall into the following three categories: supported features, unsupported features, and incompatible features. As of the 11.4 release, PIM NSR support is as follows.

PIM Supported Features. The following features are fully supported with NSR:

Auto-RP
BFD
Bootstrap router
Dense mode
Sparse mode (except for some subordinate features mentioned in the following list of unsupported features)
Source-specific multicast (SSM)
Static RPs

PIM Unsupported Features. You can configure the following PIM features on a router along with nonstop active routing, but they function as if nonstop active routing is not enabled. In other words, during a GRES event, their state information is not preserved and traffic loss is to be expected.

Internet Group Management Protocol (IGMP) exclude mode
IGMP snooping
PIM for IPv6 and related features such as embedded RP and Multicast Listener Discovery (MLD)
Policy features such as neighbor policy, bootstrap router export and import policies, scope policy, flow maps, and reverse path forwarding (RPF) check policies
Upstream assert synchronization

PIM Incompatible Features. Nonstop active routing does not support the following features, and you cannot configure them on a router enabled for PIM nonstop active routing. The commit operation fails if the configuration includes both nonstop active routing and one or more of these features:

Anycast RP
Draft-Rosen multicast VPNs (MVPNs)
Local RP
Next-generation MVPNs with PIM provider tunnels
PIM join load balancing

To work around the list of incompatible PIM features, Junos provides a configuration statement to disable NSR for PIM only. With this option, you can activate incompatible PIM features and continue to use nonstop active routing for the other protocols on the router. Before activating an incompatible PIM feature, include the `nonstop-routing disable` statement at the `[edit protocols pim]` hierarchy level. Note that in this case, nonstop active routing is disabled for all PIM features, not just the incompatible features.

NSR and RSVP-TE LSPs

Junos software extends NSR support to label-switching routers (LSR) that are part of an RSVP-TE LSP. NSR support on LSRs ensures that an LSR remains transparent to the network neighbors and that the LSP information remains unaltered during and after the switchover. You can use the `show rsvp version` command to view the NSR mode and state on an LSR. Similarly, you can use the `show mpls lsp` and `show rsvp session` commands on the standby RE to view the state that is replicated there.

As of the 11.4 Junos release, the following RSVP features are not supported for NSR:

Point-to-multipoint LSPs
Generalized Multiprotocol Label Switching (GMPLS) and LSP hierarchy
Interdomain or loose-hop expansion LSPs
BFD liveness detection

Nonstop active routing support for RSVP-TE LSPs is subject to the following limitations and restrictions:

- Control plane statistics corresponding to the `show rsvp statistics` and `show rsvp interface[detail | extensive]` commands are not maintained across RE switchovers.
- Statistics from the backup RE are not reported for show `mpls lsp statistics` and `monitor mpls label-switched-path` commands. However, if a switchover occurs, the backup RE, after taking over as the master, starts reporting statistics. Note that the `clear statistics` command issued on the old master does not have any effect on the new master, which continues to report its uncleared statistics.
- State timeouts might take additional time during nonstop active routing switchover. For example, if a switchover occurs after a neighbor has missed sending two Hello messages to the master, the new master RE waits for another three Hello periods before timing out the neighbor.
- On the RSVP ingress router, if you configure `auto-bandwidth` functionality, the bandwidth adjustment timers are set in the new master after the switchover. This causes a one-time increase in the amount of time required for the bandwidth adjustment after the switchover occurs.
- Backup LSPs—LSPs that are established between the point of local repair (PLR) and the merge point after a node or link failure—are not preserved during a RE switchover.

NSR and VRRP

Currently, VRRP does not support stateful replication, and therefore sessions with short hold times are expected to experience a VRRP mastership switch upon a GRES/SNR event. This can be avoided with a longer VRRP timer, but this workaround also increases normal failover times for non-NSR events such as a link down.

This NSR Thing Sounds Cool; So What Can Go Wrong?

Honestly? A lot.

Modern protocols are complex, and so is NSR. When a statement is made regarding NSR-support for a protocol like PIM or BGP, it’s best to try and qualify the details. Some protocol modes may be NSR supported, meaning no reset is expected, while others are NSR-compatible, which means you can commit and run the configuration but a reset is expected at NSR. In yet other cases, you may encounter a feature or protocol mode that is incompatible with NSR. In those cases, you should see a commit warning telling you where the conflict is. For example, here the operator tries to commit an L3VPN with a Rosen6-based MVPN, while NSR is enabled in 11.4. A similar error is reported for Rosen7:

```
{master}[edit]
regress@halfpint# commit check
re0:
[edit routing-instances vrf_1 protocols pim vpn-group-address]
```

```
    'vpn-group-address 239.1.1.1'
      Vpn-group-address is not supported with PIM nonstop-routing in this JunOS release.
        Vpn-group-address configuration with PIM nonstop-routing will be supported in a
        future release. To configure vpn-group-address, PIM non-stop routing must be
  disabled.
```

At least here the error message is helpful, but the result means that while waiting for MVPN NSR support in some future release you can expect PIM (in all instances) to take a hit at NSR, given you will have to disable PIM NSR support to commit this configuration, or opt to forsake MVPN support in favor of a hitless switchover. Name your poison. But, again, hitless or not, knowing what to expect at NSR can save a lot of hassles. Who has time to waste "troubleshooting" a problem and filing a defect, only to later find the feature is "working as designed"? Here, if you chose to retain MVPN you know can expect a hitless switch for unicast only; meanwhile, multicast can take several minutes of hit depending on variables such as the RP election method (static, and by extension, anycast-RP is much faster than either Auto-RP or BSR) or the timeout that's in effect on joins.

As always, with anything as complex (and critical) as a modem IP network, it's always best to consult the documentation for your release to confirm NSR support status for any specific set of features. Where possible, it's a good idea to test failover behavior with your specific configuration in effect. This allows you to spot any unexpected incompatibilities and then either redesign around the problem or adjust your network's SLAs accordingly. When making changes, always be on guard for any commit time warnings or log errors reporting replication failures or problems to make sure you stay aware of what features are or are not considered NSR-supported in any given release.

NSR, the good . . .

While trying not to sound too drunk on Juniper Kool-Aid, the capabilities described in the previous paragraphs describe what this author considers to be a truly remarkable technical feat. This author routinely tests NSR at considerable scale, in multiple dimensions, to see a hitless switchover on a PE router with 2,500 VRF EBGP Peers, 250 each OSPF and RIP peers, main instance PIM, OSPF, LDP/RSVP, COS, firewall filters, etc., combining to generate some 1.2 million active IPv4 and IPv6 routes, is, simply put, amazing.

That said, every network is different, and Junos NSR is constantly evolving to support NSR for an ever-increasing number of protocols. As such your NSR mileage may vary, significantly. The bottom line is you can see dramatically different NSR results based on the release and the specific set of protocols enabled in your network, as well as depending on whether a switchover occurs during or after the various replications mechanisms have completed their work.

. . . And the bad. As a prime example of why NSR can be frustrating to customers, consider the previously mentioned issues with PIM and the BGP MDT protocol family used to support Rosen7 multicast in a routing instance (MVPN). NSR support for PIM in a routing instance is still not present in 11.4, and this requires that you disable NSR for PIM in the main instance in order to commit a NSR configuration with MVPN. The current lack of support for PIM NSR means you can expect to lose joins and learned RP state at a NSR, which can result in several minutes of multicast traffic loss post-NSR.

In the initial v9.4 release of Rosen7, MVPN support for a new BGP protocol family known as `inet-mdt` was added to Junos. Initially, this family was not NSR supported, and the result was that any peers configured with the family would be idle (not replicated) on the BU RE. While working as designed (WAD®), the result was BGP session reset at NSR; this is particularly nasty given that the PE-PE sessions are typically configured to support other protocol families that are NSR-supported, such as `inet` or `inet-vpn`, and such connection reset results in a significant hit to the control plane, with subsequent removal of FIB entries and data loss at NSR. Note this type of outage can persist for several minutes, depending on scale; as it can only clear once, all the sessions are reestablished and the FIB is completely repopulated.

While the specific case of NSR support for `inet-mdt` was addressed sometime back, it's a good example of the dark side of NSR. Here, the difference between a completely hitless switchover, complete with requisite shock and awe, versus a significant blowup and substantial network disruption, came down to whether or not the configuration had the innocuous-looking statement needed to add `inet-mdt` support to a BGP peer.

However, even in this negative case, it bears mentioning that, unless you encounter a NSR defect, you are almost always better off with NSR than without it. In the previous case, parts of the switchover were still hitless. GRES kept the PFE from rebooting, and only the IBGP sessions that had the unsupported family were reset. This means your core IGP adjacencies, RSVP sessions, and VRF BGP sessions were maintained through the switchover, all of which helps speed the recovery of the BGP sessions that were affected, allowing the network to reconverge that much faster.

The only other options for this particular case would be to not offer MVPN services, or to opt out of NSR, perhaps in favor of a GR/GRES solution. But, as with most things in life, each workaround has its own set of drawbacks; in many cases, having to live with a known to be partially supported NSR configuration may still be your best HA option when all things are factored.

Practicing Safe NSRs

The good news is the previous MDT-related NSR incompatibility was documented, and commands existed to alert the operator to the fact that there would be disruption to BGP at switchover (i.e., the BGP connection state not being in sync between master and BU REs).

The key to knowing what to expect when your network undergoes a NSR event involves either being exceptionally well informed or in having done NSR testing in a lab environment against your specific configuration. While there are too many details and caveats for the average human to try and remember, the good news is that general principles used to verify and confirm NSR operation are not that complex.

The Preferred Way to Induce Switchovers. Or stated differently, once NSR is configured and all is up and running, how does one trigger a GRES event in a way that realistically tests the system's NSR failover behavior? This is a valid question, and the best answer is to use the CLI's `request chassis routing-engine` command to force a current master to relinquish its control or to force a current backup to seize power, the result being the same regardless of the specific form used. The operational mode CLI commands used to induce a switchover include the following:

> A `request chassis routing-engine master switch no-confirm` on the current master
>
> A `request chassis routing-engine master release no-confirm` on the current master
>
> A `request chassis routing-engine master acquire no-confirm` on the current backup

The CLI method of RE switchover is not only supported and documented, but testing has proven there are no kid gloves associated with the command, and the ensuing GRES event is as valid as any hardware-induced RE switchover, which is what the feature is designed to protect against.

Rebooting the current master when GRES is enabled is also a valid way to test NSR and GRES failover behavior. Use a `request system reboot` (or `halt`) operational mode command on the current master RE to test this method.

It's *not* recommended that you physically remove the RE from its slot, although the technique has proven popular, if not a bit therapeutic, with some customers. Yes, a switchover should happen, but in general if vandals stealing REs is the reason you network has low HA, then honestly, you have bigger issues to deal with. In rare cases, hardware damage can result from removing an RE without first performing a request system halt, which can make this kind of NSR testing unintentionally destructive.

While on the topic of what not to do when testing NSR, recall that a routing restart while under NSR is a negative test case. By this, it's meant that you should expect all sessions to flap, and that the system will eventually recover is pre-restart state. Again, a valid test case, but in no way is this how you should test for a hitless NSR; it is a valid method for GR testing, as noted previously.

Other Switchover Methods. These kernel-based switchover methods are listed for completeness' sake, as they have been known to be used for testing HA features. Remember the shell is not officially supported and hidden commands should only be used under guidance from JTAC. The following should only be considered for use in test labs when you have console access. You have been warned.

If you have access to a root shell, you can use the BSD `sysctl` function to force the kernel to panic on the master RE. As a result, the BU RE should note the lack of keepalives and assert mastership over the chassis. Note the following is done from a root shell.

```
root@Router% sysctl -w debug.kdb.panic=1
. . .

<telnet/ssh session dies>
```

At this time on the console you should see the Junos equivalent of a blue-screen-of-death as the kernel dumps core:

```
login: panic: kdb_sysctl_panic
db_log_stack_trace_cmd(c0cd4600,c0cd4600,c0c4c6dc,fbe29b80,c05ca3c4) at
  db_log_stack_trace_cmd+0x36
panic(c0c4c6dc,fbe29b8c,0,fbe29be4,1) at panic+0x264
kdb_sysctl_panic(c0c84be0,0,0,fbe29be4,fbe29be4) at kdb_sysctl_panic+0x5f
sysctl_root(fbe29be4,0,1,c0587a75,c881f700) at sysctl_root+0x134
userland_sysctl(c8789000,fbe29c54,3,0,0) at userland_sysctl+0x136
__sysctl(c8789000,fbe29cfc,18,bfbec434,fbe29d2c) at __sysctl+0xdf
syscall(fbe29d38) at syscall+0x3ce
Xint0x80_syscall() at Xint0x80_syscall+0x20
--- syscall (202, FreeBSD ELF32, __sysctl), eip = 0x88139dbb,
  esp = 0xbfbed40c, ebp = 0xbfbed438 ---
Uptime: 2m11s
Physical memory: 3571 MB
. . .

Dumping 163 MB: 148 132 116 100 84 68 52 36 20
. . .
```

The use of sysctl -w debug.kdb.panic=1 command is disabled starting in the 11.4R2 Junos release as part of security fixes to the FreeBSD kernel that were picked up via PR 723798.

```
--- JUNOS 11.4R3-S1.1 built 2012-05-18 11:03:07 UTC
. . .
root@router% sysctl -w debug.kdb.panic=1
debug.kdb.panic: 0
sysctl: debug.kdb.panic: Operation not permitted
root@router%
```

Yet another method of inducing a kernel crash is to configure the hidden debugger-on-break configuration statement and then send a break signal using the console. In most cases, this can work remotely when using a console server by using the application's send break function after you form the connection. You may need console access to the router to recover the now crashed kernel.

```
{master}[edit]
jnpr@R1-RE1# show system
debugger-on-break;
```

With the change committed and in a position to send a real break via direct serial line connection, or a Telnet break to a consoler server, you are ready to force a switchover.

Here, the latter method is used. Once connected, escape to the Telnet client's command mode. For most Unix-based command line Telnet clients, this is done with a **cnt]** sequence:

```
{master}[edit]

jnpr@R1-RE0# cnt ]
telnet>send brk
KDB: enter: Line break on console
[thread pid 11 tid 100005 ]
Stopped at      kdb_enter+0x15f:        movl    $0xc0c4c757,0(%esp)
db>
```

And again, boom goes the switchover. Now at the kernel debug prompt, you can reboot the router with a `reset` command:

```
db>reset
```

A third method of inducing a kernel-based GRES/NSR event is to generate a nonmaskable interrupt (NMI) on the current master using `sysctl` at a root shell:

```
root@Router% sysctl -w debug.induce_watchdog_timeout=1
```

A related method is to disable the system watchdog timer, which in turn raises a NMI a short while later, typically within 180 seconds:

```
root@Router% sysctl -w debug.induce_watchdog_timeout=2
```

The latter can also be achieved via the CLI by altering the configuration to disable the watchdog process, but the configuration option can be tricky to remove after testing as the ongoing watchdog timeouts will cause the kernel to reenter the debug state if you cannot roll back the configuration change quickly enough after rebooting:

```
[edit system processes]
+    watchdog disable;
```

In both cases, the NMI current master kernel to panic and enter debug mode, wherein the loss of keepalives signals the BU RE to become master inducing a GRES event.

Tips for a Hitless (and Happy) Switchover

Knowing how to induce a switchover is fine, but knowing when it's safe to do so is another matter. Here, the term safe refers to minimizing any disruption that might occur by ensuring the system is completely converged with respect to the various GRES synchronization and NSR replication tasks. On a highly scaled system, it might take 15 minutes or longer for all processes to reach a steady state in which all the GRES and NSR components have completed their work. The default 240-second GRES back-to-back switchover timer can easily expire long before a scaled system is truly ready for a successful NSR. Keep these tips in mind when testing NSR behavior; most are demonstrated and explained in detail in subsequent sections.

Wait for both master and BU REs to confirm GRES synchronization and NSR replication is complete. Use `the show task replication` and show system switchover commands on the BU and master REs, respectively, to confirm GRES synchronization and NSR replication state.

Wait for the RE to complete the download of the various next-hops into the PFE. Performing a switchover when millions of next-hops are pending installation will only delay control and dataplane convergence, given that the new master RE is pretty busy for several minutes after a switchover makes it the boss. Use the (hidden) `show krt queue` command to gauge the status of queued NH changes.

If performing a graceful restart (GR)-based GRES, be sure to wait for all routing tables to reach restart complete. Use a `show route instance detail | match pending` command to spot any table that may be waiting. GR may abort for a given instance if a GRES is performed while the instance is in a pending state.

Confirm that all IBGP and IGP sessions are in fact up/established and properly replicated on the BU RE. In some cases, unsupported options or version mismatches can leave a session unreplicated, which results in flap when the old BU becomes the new master. Consider a separate BGP session for non-NSR-supported families, like NGEN MVPN's `inet-mvpn`, to limit flap to only certain BGP sessions, thereby minimizing any resulting disruption.

Make sure that no PPMD process such as LACP or BFD are set for centralized operation (RE-based).

Run the ISSU validation check before a NSR. It can help report issues such as centralized BFD sessions or hardware that may reset at switchover. For example, stateful services PIC are generally reset during both an ISSU or NSR switchover event.

If you have any BFD sessions that are RE-based (IBGP or other types of multihop sessions), make sure the related timers can tolerate at least 15 seconds of inactivity to ensure they can survive a GRES event.

Make sure there are no silly two-second hello/six-second hold time settings for RE-based sessions such as OSPF or BGP. RE-based sessions can expect a period of from 6 to 15 seconds during which keepalives cannot be sent.

The next section details the commands and techniques used to configure and then confirm NSR in the context of BGP, IS-IS, and Layer 2 control protocols like LACP and VSTP. You can adapt the techniques shown to the protocols used in your network to confirm proper replication and that the system as a whole indicates NSR readiness, ideally before anything fails and you are forced into a switchover, ready or not! Where possible, you are always well advised to conduct actual NSR testing under your network's simulated conditions if you need to have a completely hitless NSR to meet your network's SLA guaranties. After all, educated predications can only go so far in matters of such unfathomable complexity and with the huge range of configuration variance that modern networks exhibit.

Configure NSR and NSB

For such a complicated feature, NSR and NSB are deceptively easy to configure. Much like GR, just a few global configuration statements and a router with dual REs is all you need to get going with NSR.

Before enabling NSR, you should ensure that both REs are on the same version. Again, this is not strictly necessary, but unless you are specifically directed by JTAC, or have some specific need for mismatched software versions, then having the same version on both REs is always good advice for GRES, NSR, and ISSU. In fact, the latter mandates it!

Assuming you already have the requisite `graceful-switchover` statement at the `edit chassis redundancy` hierarchy to enable GRES, then NSR and NSB can both be enabled with one statement each:

```
{master}[edit]
jnpr@R1-RE0# show routing-options
##
## Warning: Synchronized commits must be configured with nonstop routing
##
nonstop-routing;
autonomous-system 65000.65000 asdot-notation;

{master}[edit]
jnpr@R1-RE0# show protocols layer2-control
nonstop-bridging;
```

However, as noted in the `show output` warning, to commit this configuration you will need to add the `commit synchronize` option to the configuration. Unlike GRES, which *encourages* you to synchronize the configs across both REs at each commit, NSR mandates it, and this is enforced through the set `system commit synchronize` option:

```
{master}[edit]
jnpr@R1-RE0# set system commit synchronize
```

The warning is now removed and the NSR/NSB configuration can be committed:

```
{master}[edit]
jnpr@R1-RE0# show routing-options
nonstop-routing;
autonomous-system 65000.65000 asdot-notation;

{master}[edit]
jnpr@R1-RE0# commit
re0:
. . .
```

Note that you cannot commit a NSR configuration if you have `graceful-restart` in effect, for reasons described in the following.

NSR and Graceful Restart: Not like Peanut Butter and Chocolate

Users are often surprised to learn that Junos does not allow you to configure both GR and NSR at the same time. I mean, why not have your cake and grow fat while eating it too? The simple answer is “because you can’t.” This is more than just a simple Junos limitation, as the two HA approaches are somewhat diametrically opposed in how they do their business. Think about it: GR open admits the control plane restart and expects neighbors to help it recover, while the goal of NSR is to have no externally visible indication of a restart. You simply can’t do both at the same time. However, enabling NSR only prevents GR restart modes. By default, most protocols support GR helper mode and can therefore assist a GR-configured neighbor through its restart even though locally NSR is configured.

> One big difference between GR and NSR is the ability to perform a hitless restart routing on the local RE. The former supports a routing restart as well as GRES-based failover testing, while the latter can only be tested with GRES events. The bottom line is when NSR is running, a restart routing causes all sessions to flap, which is, of course, the polar opposite of hitless.

General NSR Debugging Tips

NSR and RE switchovers in a network with multiple complex protocols running, at scale, can be a marvelous thing to behold when all goes to plan, and a daunting task to troubleshoot when things don’t. When troubleshooting NSR issues, keep the following tips in mind:

Confirm protocol state on master and backup REs. Mismatched state normally spells a problem at switchover. Generally speaking, aside from flap count and up time, you expect the same state for all NSR-supported protocols.

Use protocol-specific replication commands to help identify and troubleshoot replication problems. Some of these commands are hidden but most are documented. Note that unlike protocol state, many of these replication-related commands return different results depending on whether you execute the command on the master or a BU RE.

Always confirm overall system-level GRES and replication state using the `show system switchover command` on the BU RE and the `show task replication` command on the master, respectively, before requesting a switchover. Note that on a highly scaled system, it can take several minutes for replication to even begin after a GRES and upwards of 10 or more minutes for all protocols to complete replication. The 240-second back-to-back GRES hold-down time can easily expire before a scaled system has completed replication from a previous switchover.

Watch out for any statements that disable distributed mode `ppmd` clients such as LACP or BFD, and know whether any of your sessions are RE based, and if so, be sure to set timeouts that are longer than the RE blackout during the switchover.

On MX routers with 11.4, you should assume the blackout window can last 7.5 seconds and maybe even longer on highly scaled systems.

Junos tracing is always a useful tool when troubleshooting. Don't forget to add NSR-, GRES-, and replication-related trace flags to any protocol tracing to help spot issues relating to NSR. Also, make sure you look at both the old, preswitchover logs on the new backup, as well as the post-NSR logs on the new master to make sure you get the full picture. Tracing NSR for the first time, when you suspect there is a problem, is a good way to get mislead and wind up filing a mistaken problem report (PR). Whenever possible, its best to trace successful NSR events so you have a baseline as to what is normal.

Try and isolate to the lowest layer that is showing an unexpected flap. For example, if you are running a L3VPN service over LDP that's in turn tunneled over RSVP signaled LSPs, with BFD protection on IGP sessions, then just about anything going wrong can be expected to disrupt the L3VPN service. Try and find the first layer that flaps (e.g., OSPF for the sake of argument) and then ignore protocols and services that ride on top such as IBGP and RSVP, at least until the OSPF issue is sorted.

Verify NSR and NSB

Figure 9-10 shows the test topology used for verification of NSR and NSB.

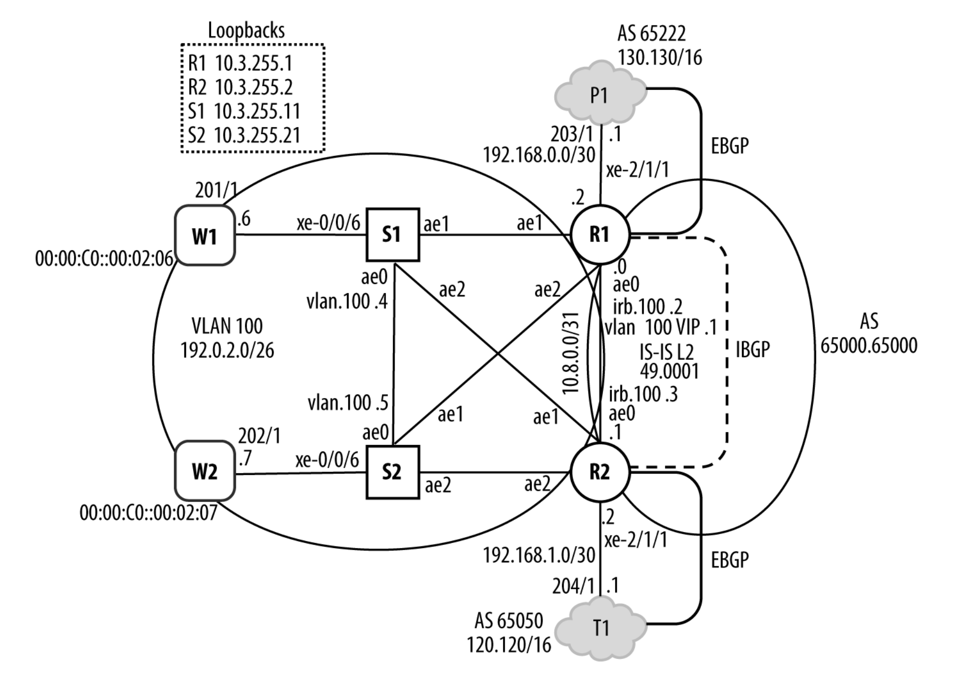


Figure 9-10. Nonstop Routing and Bridging Test Topology.

The test topology is based on the standard “hybrid L2/L3” data center design, as this affords the opportunity to test and explore both routing and bridging operation through an NSR event. The setup has been modified to place VLAN 100 into effect at S2’s xe-0/0/6 interface so that Layer 2 test traffic can be sent through VLAN 100 using router tester ports 201/1 and 201/2. The tester ports are assigned .6 and .7 host identifiers from the 192.0.2.0/26 LIS associated with VLAN 100; the VLAN 100-related IP addresses assigned to the VLAN and IRB interfaces at both switches and routers are also shown. The MAC addresses of both router tester ports are documented, as this information is relevant to the Layer 2 domain’s learning and forwarding operation. Note that while VLAN 200 is still provisioned, with R2 still the VSTP root for that VLAN as before, in this example we focus only on VLAN 100 and the backbone routing behavior through an NSR-based GRES event allowing us to omit VLAN 200 details from the figure.

In this example, IS-IS level 2 is operating as the IGP between R1 and R2 over the ae0.1 link that serves as the network’s Layer 3 backbone. IBGP peering using a 32-bit ASN has been established between the router’s lo0 addresses, as per best practices; in like fashion, interface-based EBGP peering is in effect to external peers P1 and T1, who advertise routes in the 130.130/16 and 120.120/16 ranges, respectively. Three aggregate routes are defined at R1 and R2 that encompass the loopback, backbone, and

VLAN address space, along with a simple export policy at both routers to advertise all three of the aggregates to their external peers.

The routing options and policy at R1 is shown:

```
{master}[edit]
jnpr@R1-RE0# show routing-options
nonstop-routing;
aggregate {
    route 192.0.2.0/25;
    route 10.8.0.0/24;
    route 10.3.255.0/24;
}
autonomous-system 65000.65000 asdot-notation;

{master}[edit]
jnpr@R1-RE0# show policy-options
policy-statement bgp_export {
    term 1 {
        from protocol aggregate;
        then accept;
    }
}
```

Again, note the use of a 32-bit ASN, which is becoming common in Enterprises due to lack of ASN space in the 16-bit format. Here the `asdot-notation` switch causes such an AS to be displayed as configured (i.e., 65000.65000, as opposed to the 4259905000 that would otherwise be shown). Moving on, the protocols stanza is shown, again at R1; here the focus is on IS-IS and BGP, but the VSTP-related configuration is also shown:

```
{master}[edit]
jnpr@R1-RE0# show protocols
bgp {
    path-selection external-router-id;
    log-updown;
    group p1 {
        type external;
        export bgp_export;
        peer-as 65222;
        neighbor 192.168.0.1;
    }
    group int {
        type internal;
        local-address 10.3.255.1;
        family inet {
            unicast;
        }
        bfd-liveness-detection {
            minimum-interval 150;
            multiplier 3;
        }
        neighbor 10.3.255.2;
    }
}
```

```
    isis {
        reference-bandwidth 100g;
        level 1 disable;
        interface xe-2/1/1.0 {
            passive;
        }
        interface ae0.1 {
            point-to-point;
            bfd-liveness-detection {
                minimum-interval 150;
                multiplier 3;
            }
        }
        interface lo0.0 {
            passive;
        }
    }
    lldp {
        interface all;
    }
    layer2-control {
        nonstop-bridging;
    }
    vstp {
        interface xe-0/0/6;
        interface ae0;
        interface ae1;
        interface ae2;
        vlan 100 {
            bridge-priority 4k;
            interface xe-0/0/6;
            interface ae0;
            interface ae1;
            interface ae2;
        }
        vlan 200 {
            bridge-priority 8k;
            interface ae0;
            interface ae1;
            interface ae2;
        }
    }
```

The example makes use of BFD session protection for both the IS-IS adjacency and the IBGP session between R1 and R2, both using 150 ms as the minimum interval with a multiplier of three. It's not typical to see BFD protection for an IBGP session. The configuration and functionality is supported in Junos, and the reason for this configuration will become clear a bit later on.

> The use of BFD to protect IBGP sessions is not a current best practice. Typically, IBGP is multihop loopback-based, and therefore benefits from the IGP's ability to reroute around failure while keeping the BGP session alive as long as there is a restoration of connectivity within the BGP session's hold-timer, which is normally rather long at 90 seconds. Adding BFD to an IBGP session results in session teardown basic on the typically short duration BFD timer settings, which is a behavior that is at odds with IBGP stability during an IGP reconvergance event. It's better practice to confine BFD to the IGP sessions, which in turn helps the IGP detect faults and reconverge faster while leaving IBGP to its hold timer.
>
> Because EBGP is often based on direct interface peering that does not require a IGP, the use of BFD to protect single-hop EBGP sessions is reasonable.

Note the BGP stanza includes the path selection statement to ensure the same active route decision is made by both the active and standby REs, as described previously. Also note how passive IS-IS is specified to run on the EBGP links; this is a common approach to solving EBGP next-hop reachability. The passive setting ensures no adjacency can form while still allowing IS-IS to advertise the related IP subnet as an internal route. The other common approach here is a *next-hop-self* policy to overwrite the external NH with the IBGP speaker's lo0 address, which again is reachable as an internal IS-IS route.

Also of note is the *absence* of a LACP stanza, showing that the default distributed mode is in effect. In distributed mode, the session keepalive functionality is distributed into the PFE, which is extremely important for a successful NSR; if you use the `ppm centralized` statement for LACP, as shown in the following, you will get LACP flap at GRES, which in turn nets a less than desirable NSR result as generally speaking, any protocols that ride over the affected interface will see the transition and follow shortly thereafter with a flap of their own:

```
{master}[edit]
jnpr@R1-RE0# show protocols lacp
ppm centralized;
```

Confirm Pre-NSR Protocol State

Before doing anything NSR-specific, we quickly access the steady state of the network. BGP and IS-IS is confirmed:

```
{master}[edit]
jnpr@R1-RE0# run show isis adjacency
Interface             System         L State        Hold (secs) SNPA
ae0.1                 R2-RE0         2  Up                   21

{master}[edit]
jnpr@R1-RE0# run show route protocol isis
```

```
inet.0: 219 destinations, 219 routes (219 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

10.3.255.2/32      *[IS-IS/18] 20:16:56, metric 5
> to 10.8.0.1 via ae0.1
192.168.1.0/30     *[IS-IS/18] 00:29:02, metric 15
> to 10.8.0.1 via ae0.1
{master}[edit]
jnpr@R1-RE0# run show bgp summary
Groups: 2 Peers: 2 Down peers: 0
Table          Tot Paths  Act Paths Suppressed    History Damp State    Pending
inet.0               200        200          0          0          0          0
Peer                     AS      InPkt     OutPkt    OutQ   Flaps Last Up/Dwn State|
  #Active/Received/Accepted/Damped...
10.3.255.2      65000.65000         67         68       0       0
  29:04 100/100/100/0        0/0/0/0
192.168.0.1           65222         59         74       0       0
  29:08 100/100/100/0        0/0/0/0

{master}[edit]
jnpr@R1-RE0# run show route protocol bgp

inet.0: 219 destinations, 219 routes (219 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

120.120.0.0/24     *[BGP/170] 00:29:13, localpref 100, from 10.3.255.2
                      AS path: 65050 ?
> to 10.8.0.1 via ae0.1
120.120.1.0/24     *[BGP/170] 00:29:13, localpref 100, from 10.3.255.2
                      AS path: 65050 ?
> to 10.8.0.1 via ae0.1
120.120.2.0/24     *[BGP/170] 00:29:13, localpref 100, from 10.3.255.2
                      AS path: 65050 ?
. . .
```

BGP and IS-IS are as expected, so we quickly look at Layer 2 protocols and functions. A quick look at BFD:

```
{master}[edit]
jnpr@R1-RE0# run show bfd session
                                                  Detect   Transmit
Address                  State     Interface      Time     Interval  Multiplier
10.3.255.2               Up                       0.450     0.150        3
10.8.0.1                 Up        ae0.1          0.450     0.150        3

2 sessions, 2 clients
Cumulative transmit rate 7.7 pps, cumulative receive rate 7.7 pps
```

And now LACP:

```
{master}[edit]
jnpr@R1-RE0# run show lacp interfaces
Aggregated interface: ae0
    LACP state:       Role   Exp   Def  Dist  Col  Syn  Aggr  Timeout  Activity
      xe-2/0/0       Actor    No    No   Yes  Yes  Yes   Yes     Fast    Active
      xe-2/0/0      Partner    No     No    Yes  Yes  Yes   Yes      Fast    Active
      xe-2/0/1        Actor    No     No    Yes  Yes  Yes   Yes      Fast    Active
      xe-2/0/1      Partner    No     No    Yes  Yes  Yes   Yes      Fast    Active
    LACP protocol:        Receive State  Transmit State          Mux State
      xe-2/0/0                  Current   Fast periodic Collecting distributing
      xe-2/0/1                  Current   Fast periodic Collecting distributing

Aggregated interface: ae1
    LACP state:       Role   Exp   Def  Dist  Col  Syn  Aggr  Timeout  Activity
      xe-2/2/0        Actor    No     No    Yes  Yes  Yes   Yes      Fast    Active
      xe-2/2/0      Partner    No     No    Yes  Yes  Yes   Yes      Fast    Active
      xe-2/2/1        Actor    No     No    Yes  Yes  Yes   Yes      Fast    Active
      xe-2/2/1      Partner    No     No    Yes  Yes  Yes   Yes      Fast    Active
    LACP protocol:        Receive State  Transmit State          Mux State
      xe-2/2/0                  Current   Fast periodic Collecting distributing
      xe-2/2/1                  Current   Fast periodic Collecting distributing

Aggregated interface: ae2
    LACP state:       Role   Exp   Def  Dist  Col  Syn  Aggr  Timeout  Activity
      xe-2/3/0        Actor    No     No    Yes  Yes  Yes   Yes      Fast    Active
      xe-2/3/0      Partner    No     No    Yes  Yes  Yes   Yes      Fast    Active
      xe-2/3/1        Actor    No     No    Yes  Yes  Yes   Yes      Fast    Active
      xe-2/3/1      Partner    No     No    Yes  Yes  Yes   Yes      Fast    Active
    LACP protocol:        Receive State  Transmit State          Mux State
      xe-2/3/0                  Current   Fast periodic Collecting distributing
      xe-2/3/1                  Current   Fast periodic Collecting distributing
```

As lastly, VSTP status for VLAN 100:

```
jnpr@R1-RE0# run show spanning-tree interface vlan-id 100

Spanning tree interface parameters for VLAN 100

Interface    Port ID    Designated      Designated         Port    State  Role
                         port ID        bridge ID          Cost
ae0            128:483      128:483   4196.001f12b88fd0     1000  FWD    DESG
ae1            128:484      128:484   4196.001f12b88fd0     1000  FWD    DESG
ae2            128:485      128:485   4196.001f12b88fd0     1000  FWD    DESG
```

The Layer 2 and 3 control plane state are as expected. Data plane stimulation is started by sending bidirectional Layer 2 and Layer 3 streams over VLAN 100 and between the EBGP peers, respectively. All four streams generate 128-byte IP packets at a constant rate. The Layer 2 streams are at 80% line rate (based on 10GE), whereas the Layer 3 streams are at 85% so they can be tracked separately on the tester's throughput graphs. The Layer 2 streams are sent to (and from) the MAC and IP addresses of the 201/1 and 202/1 tester ports; the presence of the bidirectional flows allows these MACs to be learned so that we avoid unknown unicast flooding. The Layer 3 stream is sent to (and from) the second EBGP route in each of the EBGP route pools, which is to say 130.130.1.1 and 120.120.1.1, respectively. The relatively high traffic rates and use of a N2X router tester helps confirm that claims of hitless dataplane (and control plane) operation are justified; after all, this is not your grandfather's ping testing, which, honestly, is not a very good way to test for data plane behavior in today's high-speed world.

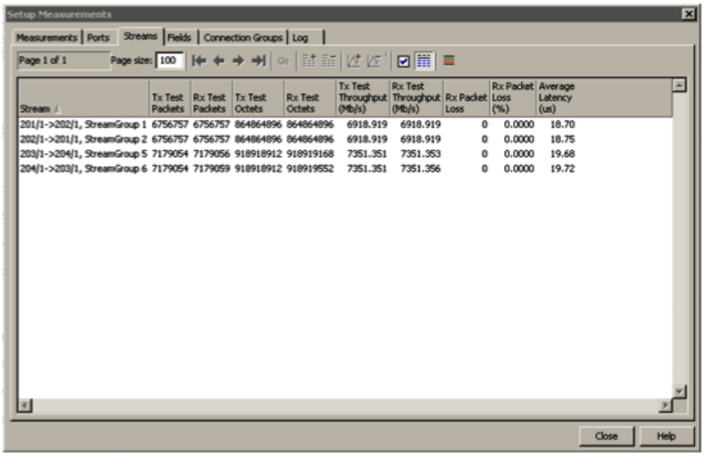
The `monitor interface` command is used to quickly confirm traffic volumes on key interfaces. At R1, the ae1 interface carries the Layer 2 domain's traffic:

```
Next='n', Quit='q' or ESC, Freeze='f', Thaw='t', Clear='c', Interface='i'
R1-RE0                            Seconds: 0                   Time: 11:10:18
                                                           Delay: 4/4/4
Interface: ae1, Enabled, Link is Up
Encapsulation: Ethernet, Speed: 20000mbps
Traffic statistics:                                           Current delta
  Input bytes:            72440901925378 (6919050752 bps)               [0]
  Output bytes:           35921675228538 (6919051008 bps)               [0]
  Input packets:            287143751453 (6756885 pps)                  [0]
  Output packets:           144496826189 (6756887 pps)                  [0]
Error statistics:
  Input errors:                        0                                [0]
  Input drops:                         0                                [0]
  Input framing errors:                0                                [0]
  Carrier transitions:                 0                                [0]
  Output errors:                       0                                [0]
  Output drops:                        0                                [0]
```

While ae0 is transporting the BGP-based traffic:

```
Next='n', Quit='q' or ESC, Freeze='f', Thaw='t', Clear='c', Interface='i'
R1-RE0                            Seconds: 1                   Time: 11:20:05
                                                           Delay: 0/0/0
Interface: ae0, Enabled, Link is Up
Encapsulation: Ethernet, Speed: 20000mbps
Traffic statistics:                                           Current delta
  Input bytes:             6893931459049 (6317555336 bps)               [0]
  Output bytes:           35708294654929 (6317555224 bps)               [0]
  Input packets:             36863464745 (7179044 pps)                  [0]
  Output packets:           149832027353 (7179043 pps)                  [0]
Error statistics:
  Input errors:                        0                                [0]
  Input drops:                         0                                [0]
  Input framing errors:                0                                [0]
  Carrier transitions:                 0                                [0]
  Output errors:                       0                                [0]
  Output drops:                        0                                [0]
```

The reported traffic loads on R1's AE interfaces correspond nicely with the router tester displays, as shown in Figure 9-11, and confirm flow symmetry in both the Layer 2 and Layer 3 flows.



Stream	Tx Test Packets	Rx Test Packets	Tx Test Octets	Rx Test Octets	Tx Test Throughput (Mb/s)	Rx Test Throughput (Mb/s)	Rx Packet Loss	Rx Packet Loss (%)	Average Latency (us)
201/1->202/1, StreamGroup 1	6756757	6756757	864864896	864864896	6918.919	6918.919	0	0.0000	18.70
202/1->201/1, StreamGroup 2	6756757	6756757	864864896	864864896	6918.919	6918.919	0	0.0000	18.75
203/1->204/1, StreamGroup 5	7179054	7179056	918918912	918919168	7351.351	7351.353	0	0.0000	19.68
204/1->203/1, StreamGroup 6	7179054	7179059	918918912	918919552	7351.351	7351.356	0	0.0000	19.72

Figure 9-11. Pre-NSR (Instantaneous) Traffic Statistics

With the initial state confirmed, attention shifts to confirmation of NSR and NSB replication, as detailed in the next section.

Confirm Pre-NSR Replication State

We begin preswitchover confirmation with GRES readiness as it's a prerequisite to a successful NSR switchover.

```
{backup}
jnpr@R1-RE1>show system switchover
Graceful switchover: On
Configuration database: Ready
Kernel database: Ready
Peer state: Steady State
```

The BU RE at R1 confirms that GRES replication has completed successfully, an auspicious first sign. Overall replication is now verified on the master RE:

```
{master}
jnpr@R1-RE0>show task replication
        Stateful Replication: Enabled
        RE mode: Master

    Protocol                Synchronization Status
    BGP                     Complete
    IS-IS                   Complete

{master}
jnpr@R1-RE0>
```

Again, the output is as expected; you only expect replication status for NSR-supported Layer 3 protocols, and here there are two such protocols running, namely IS-IS and BGP.

BGP Replication. BGP replication offers specific details through CLI show commands:

```
{master}
jnpr@R1-RE0>show bgp re?
Possible completions:
  replication          BGP NSR replication state between master and backup
```

Details for BGP on the master RE are shown:

```
jnpr@R1-RE0>show bgp replication
Synchronization master:
  Session state: Up, Since: 2:03:47
  Flaps: 0
  Protocol state: Idle, Since: 2:03:47
  Synchronization state: Complete
  Number of peers waiting: AckWait: 0, SoWait: 0, Scheduled: 0
  Messages sent: Open 1, Establish 2, Update 0, Error 0, Complete 1
  Messages received: Open 1, Request 1 wildcard 0 targeted, EstablishAck 2,
    CompleteAck 1
```

The key information in the BGP replication displays is the confirmation of an established session with 0 flaps, which shows the expected stability of the replication process. Also good is the lack of queued messages pending, and the 0 error count. In summary, the display confirms that BGP replication has completed with no errors and is stable, just what you want to see. The same command can be run on the BU RE; while the output is relatively terse, no errors are reported:

```
{backup}
jnpr@R1-RE1>show bgp replication
Synchronization backup:
 State: Established 2:04:21 ago
```

Given that all BGP replication appears to have completed normally, you expect to find matching state between the REs for both BGP peers and routes. The master view of overall BGP operation is displayed first:

```
{master}
jnpr@R1-RE0>show bgp summary
Groups: 2 Peers: 2 Down peers: 0
Table          Tot Paths  Act Paths Suppressed    History Damp State    Pending
inet.0               200        200          0          0          0          0
Peer                     AS      InPkt     OutPkt    OutQ   Flaps Last
   Up/Dwn State|#Active/Received/Accepted/Damped...
10.3.255.2      65000.65000        289        289       0       0
  2:09:28 100/100/100/0        0/0/0/0
192.168.0.1           65222        260        295       0       0
  2:09:32 100/100/100/0        0/0/0/
```

And the same on the BU:

```
{backup}
jnpr@R1-RE1>show bgp summary
Groups: 2 Peers: 2 Down peers: 0
Table          Tot Paths  Act Paths Suppressed    History Damp State    Pending
inet.0               200        200          0          0          0          0
Peer                     AS      InPkt     OutPkt    OutQ   Flaps Last Up/Dwn
```

```
  State|#Active/Received/Accepted/Damped...
10.3.255.2       65000.65000        290        289     100       0
  2:09:52 100/100/100/0       0/0/0/0
192.168.0.1            65222        260        294     103       0
  2:09:56 100/100/100/0       0/0/0/0
```

The master's view of the external peer:

```
{master}
jnpr@R1-RE0>show bgp neighbor 192.168.0.1
Peer: 192.168.0.1+179 AS 65222 Local: 192.168.0.2+56140 AS 65000.65000
  Type: External    State: Established    Flags: <Sync RSync>
  Last State: EstabSync     Last Event: RecvKeepAlive
  Last Error: None
  Export: [ bgp_export ]
  Options: <Preference LogUpDown PeerAS Refresh>
  Holdtime: 90 Preference: 170
  Number of flaps: 0
  Peer ID: 192.168.0.1     Local ID: 10.3.255.1       Active Holdtime: 90
  Keepalive Interval: 30         Peer index: 0
  BFD: disabled, down
  Local Interface: xe-2/1/1.0
  NLRI for restart configured on peer: inet-unicast
  NLRI advertised by peer: inet-unicast
  NLRI for this session: inet-unicast
  Peer does not support Refresh capability
  Stale routes from peer are kept for: 300
  Peer does not support Restarter functionality
  Peer does not support Receiver functionality
  Peer does not support 4 byte AS extension
  Peer does not support Addpath
  Table inet.0 Bit: 10000
    RIB State: BGP restart is complete
    Send state: in sync
    Active prefixes:              100
    Received prefixes:            100
    Accepted prefixes:            100
    Suppressed due to damping:    0
    Advertised prefixes:          103
  Last traffic (seconds): Received 2    Sent 17    Checked 62
  Input messages:  Total 263    Updates 1       Refreshes 0     Octets 5419
  Output messages: Total 298    Updates 7       Refreshes 0     Octets 6402
  Output Queue[0]: 0
  Trace options: graceful-restart
  Trace file: /var/log/bgp_trace size 10485760 files 1
```

And on the BU:

```
{backup}
jnpr@R1-RE1>show bgp neighbor 192.168.0.1
Peer: 192.168.0.1 AS 65222     Local: 192.168.0.2 AS 65000.65000
  Type: External    State: Established    Flags: <ImportEval Sync>
  Last State: Idle          Last Event: RecvEstab
  Last Error: None
  Export: [ bgp_export ]
  Options: <Preference LogUpDown PeerAS Refresh>
  Holdtime: 90 Preference: 170
  Number of flaps: 0
  Peer ID: 192.168.0.1     Local ID: 10.3.255.1       Active Holdtime: 90
  Keepalive Interval: 30        Peer index: 0
  BFD: disabled, down
  Local Interface: xe-2/1/1.0
  NLRI for restart configured on peer: inet-unicast
  NLRI advertised by peer: inet-unicast
  NLRI for this session: inet-unicast
  Peer does not support Refresh capability
  Peer does not support Restarter functionality
  Peer does not support Receiver functionality
  Peer does not support 4 byte AS extension
  Peer does not support Addpath
  Table inet.0 Bit: 10000
    RIB State: BGP restart is complete
    Send state: in sync
    Active prefixes:              100
    Received prefixes:            100
    Accepted prefixes:            100
    Suppressed due to damping:    0
    Advertised prefixes:          103
  Last traffic (seconds): Received 26   Sent 13   Checked 139043
  Input messages:  Total 263    Updates 1       Refreshes 0     Octets 5419
  Output messages: Total 297    Updates 7       Refreshes 0     Octets 6343
  Output Queue[0]: 103
  Trace options: graceful-restart
  Trace file: /var/log/bgp_trace size 10485760 files 10
```

And last, a specific BGP route on the master:

```
{master}
jnpr@R1-RE0>show route 130.130.1.1

inet.0: 219 destinations, 219 routes (219 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

130.130.1.0/24     *[BGP/170] 02:13:09, localpref 100
                      AS path: 65222 ?
> to 192.168.0.1 via xe-2/1/1.0
```

And again on the BU:

```
{backup}
jnpr@R1-RE1>show route 130.130.1.1

inet.0: 219 destinations, 426 routes (219 active, 0 holddown, 207 hidden)
+ = Active Route, - = Last Active, * = Both

130.130.1.0/24     *[BGP/170] 02:12:46, localpref 100
                      AS path: 65222 ?
> to 192.168.0.1 via xe-2/1/1.0
```

At this point, you should be getting the idea with regard to seeing matching state for supported protocols between REs, so the point is not belabored farther.

IS-IS Replication. IS-IS replication commands are currently hidden in the CLI, likely due to a lack of documentation and the belief that customers should not bother themselves with such details.

```
{master}
jnpr@R1-RE0>show isis repl?
No valid completions
{master}
jnpr@R1-RE0> show isis repl
```

Completing the hidden argument allows the options to be viewed:

```
{master}
jnpr@R1-RE0>show isis replication ?
Possible completions:
  adjacency              Show IS-IS adjacency replication information
  database               Show IS-IS link-state database replication information
  interface              Show IS-IS interface replication information
  statistics             Show IS-IS replication statistics
  system-id              Show IS-IS system-id replication information
{master}
jnpr@R1-RE0> show isis replication
```

A few of the hidden IS-IS replication command options are explored, again on both master and BU, as always, starting with the current master:

```
{master}
jnpr@R1-RE0>show isis replication adjacency
IS-IS adjacency replication:
Interface             System         L State            SNPA
ae0.1                 R2-RE0         1 Up               0:1f:12:b7:df:c0
  Instance: master, Interface index: 325, References: 1
  Up/Down transitions: 3, 22:18:23 ago
  Last event: Seenself

{master}
jnpr@R1-RE0>show isis replication database
IS-IS level 1 link-state database replication:
Instance: master
  0 LSP Replication Entries

IS-IS level 2 link-state database replication:
Instance: master
LSP ID                      Sequence Checksum Lifetime Used
R1-RE0.00-00                   0x40b   0x53af     1092  Yes
R2-RE0.00-00                   0x38b    0x974      880  Yes
  2 LSP Replication Entries
```

And now the same on the BU:

```
{backup}
jnpr@R1-RE1>show isis replication adjacency
IS-IS adjacency replication:
Interface             System         L State            SNPA
ae0.1                 R2-RE0         1 Up               0:1f:12:b7:df:c0
  Instance: master, Interface index: 325, References: 1
```

```
    Up/Down transitions: 3, 22:19:29 ago
    Last event: Seenself

  {backup}
  jnpr@R1-RE1>show isis replication database
  IS-IS level 1 link-state database replication:
  Instance: master
    0 LSP Replication Entries

  IS-IS level 2 link-state database replication:
  Instance: master
  LSP ID                      Sequence Checksum Lifetime Used
  R1-RE1.00-00                   0x40b    0x53af     1022  Yes
  R2-RE0.00-00                   0x38b     0x974      811  Yes
    2 LSP Replication Entries
```

The various (hidden) command displays for IS-IS replication are as expected. As with BGP, the bottom line for IS-IS is the same as for any NSR-supported protocol: you expect matching displays for the various show commands on both the REs. A few IS-IS operational commands are executed on both master and BU RE to confirm, first on the master:

```
{master}
jnpr@R1-RE0>show isis adjacency
Interface             System         L State        Hold (secs) SNPA
ae0.1                 R2-RE0         2  Up                   22

{master}
jnpr@R1-RE0>show isis route
 IS-IS routing table             Current version: L1: 0 L2: 112
IPv4/IPv6 Routes
----------------
Prefix             L Version   Metric Type Interface       NH   Via
10.3.255.2/32      2     112        5 int  ae0.1           IPV4 R2-RE0
192.168.1.0/30     2     112       15 int  ae0.1           IPV4 R2-RE0
```

And again on the BU:

```
{backup}
jnpr@R1-RE1>show isis adjacency
Interface             System         L State        Hold (secs) SNPA
ae0.1                 R2-RE0         2  Up                    0

{backup}
jnpr@R1-RE1>show isis route
 IS-IS routing table             Current version: L1: 0 L2: 84
IPv4/IPv6 Routes
----------------
Prefix             L Version   Metric Type Interface       NH   Via
10.3.255.2/32      2      84        5 int  ae0.1           IPV4 R2-RE0
192.168.1.0/30     2      84       15 int  ae0.1           IPV4 R2-RE0
```

Confirm BFD Replication. While not a Layer 3 protocol, in this case BFD is used to protect both IS-IS and BGP, so NSR confirmation is covered in this section. Things start with confirmation of session status and their replication state on both REs. Recall that in theory these sessions have their hellos distributed into the PFE, where they are handled by the periodic packet management daemon (PPM), thus accommodating the relatively short detection times that are in effect through a NSR:

```
jnpr@R1-RE0>show bfd session detail
                                                  Detect   Transmit
Address                  State     Interface      Time     Interval  Multiplier
10.3.255.2               Up                       0.450     0.150        3
 Client BGP, TX interval 0.150, RX interval 0.150
 Session up time 03:41:43
 Local diagnostic None, remote diagnostic None
 Remote state Up, version 1
 Replicated
                                                  Detect   Transmit
Address                  State     Interface      Time     Interval  Multiplier
10.8.0.1                 Up        ae0.1          0.450     0.150        3
 Client ISIS L2, TX interval 0.150, RX interval 0.150
 Session up time 23:40:32
 Local diagnostic None, remote diagnostic None
 Remote state Up, version 1
 Replicated

2 sessions, 2 clients
Cumulative transmit rate 13.3 pps, cumulative receive rate 13.3 pps
```

And on the BU:

```
{backup}
jnpr@R1-RE1>show bfd session detail
                                                  Detect   Transmit
Address                  State     Interface      Time     Interval  Multiplier
10.3.255.2               Up                       0.450     0.150        3
 Client BGP, TX interval 0.150, RX interval 0.150
 Session up time 03:42:18
 Local diagnostic None, remote diagnostic None
 Remote state Up, version 1
 Replicated
                                                  Detect   Transmit
Address                  State     Interface      Time     Interval  Multiplier
10.8.0.1                 Up        ae0.1          0.450     0.150        3
 Client ISIS L2, TX interval 0.150, RX interval 0.150
 Session up time 23:41:07
 Local diagnostic None, remote diagnostic None
 Remote state Up, version 1
 Replicated

2 sessions, 2 clients
Cumulative transmit rate 13.3 pps, cumulative receive rate 13.3 pps
```

Note how the BGP-related BFD session, being multihop as the related session is formed between router loopbacks, does not show an associated interface; the ability to survive the failure of a network interface is the whole point of loopback-based peering, after all. Once again, replication-specific commands are currently hidden for BFD, so be sure to type out the complete `replication` keyword if you plan to use as part of NSR troubleshooting.

```
{master}
jnpr@R1-RE0>show bfd replication ?
Possible completions:
  queue                Show data-mirroring queues
  registered-database  Show registered databases for mirroring
  session              Show session replication database
  statistics           Show replication statistics
{master}
```

A few of the BFD replication commands are executed on the master:

```
{master}
jnpr@R1-RE0>show bfd replication statistics
Connection resets: 1
Last connection close: At Tue Feb 14 20:02:42 2012
Last connection close:  mirror_connect_peer:2009 errno 0
Database resyncs: 1
Bytes sent: 33832
Bytes received: 25768

{master}
jnpr@R1-RE0>show bfd replication session
Address                    Interface     Discriminator  Replication state
10.8.0.1                   ae0.1                     6       Synchronized
10.3.255.2                                           9       Synchronized
```

And now the BU:

```
{backup}
jnpr@R1-RE1>show bfd replication statistics
Connection resets: 0
Last connection close: At Tue Feb 14 20:02:44 2012
Last connection close:  mirror_peer_connect_complete:1971 errno 0
Database resyncs: 0
Bytes sent: 25848
Bytes received: 33912

{backup}
jnpr@R1-RE1>show bfd replication session
Address                    Interface     Discriminator  Replication state
10.8.0.1                   ae0.1                     6             Target
10.3.255.2                                           9             Target
```

All indications are that BFD has completed replication normally with no issues or errors that should jeopardize a NSR event. In fact, at this point all operational indications are that Layer 3 NSR, in this case for IS-IS, BFD, and BGP, is working as designed. Both GRES synchronization and NSR protocol-based replication have completed, and the system appears NSR ready. At this stage, attention shifts to preswitchover confirmation of NSB, as detailed in the following.

Layer 2 NSB Verification. When performing NSB verification on the MX router, at least in the 11.4 release, it's pretty clear that different teams within Juniper worked on the NSR and NSB feature sets, and again, between the MX and EX implementation of NSB. This isn't too surprising given the different maturity levels and the differences in hardware architecture between the EX and MX platforms.

NSB is confirmed by the presence of the `l2cpd` running on both REs. Starting at the master:

```
{master}
jnpr@R1-RE0>show l2cpd ?
Possible completions:
  task                 Show l2cpd per-task information
{master}
jnpr@R1-RE0> show l2cpd task
Pri Task Name                           Pro  Port So Flags
 15 Memory
 20 RT
 40 FNP                                           18 <>
 40 MRPTXRX                                       17 <>
 40 LLDP IO                              11       16 <>
 40 LLDPD_IF
 40 l2cpd issu
 40 Per IFD Feature                               15 <>
 40 PNACTXRX                                      14 <>
 40 PNACAUTH                                      13 <Connect>
 40 PNACD
 40 STPD
 40 ERPD
 40 STP I/O./var/run/ppmd_control                  8 <>
 41 MRPD
 50 MVRP l2ald ipc./var/run/l2ald_control
                                                  25 <>
 50 L2CPD Filter
 60 Mirror Task.8.1.c3.a0.80.0.0.6                29 <>
 60 knl Ifstate                                    6 <>
 60 KNL
 70 MGMT.local                                    27 <>
 70 MGMT_Listen./var/run/l2cpd_mgmt               24 <Accept>
 70 SNMP Subagent./var/run/snmpd_agentx           26 <>
```

And now on the BU:

```
{backup}
jnpr@R1-RE1>show l2cpd task
Pri Task Name                           Pro  Port So Flags
 15 Memory
 20 RT
 40 FNP                                           19 <>
 40 MRPTXRX                                       18 <>
 40 LLDP IO                              11       17 <>
 40 LLDPD_IF
 40 l2cpd issu
 40 Per IFD Feature                               16 <>
 40 PNACTXRX                                      15 <>
 40 PNACAUTH                                      14 <Connect>
 40 PNACD
 40 STPD
```

```
40 ERPD
40 STP I/O./var/run/ppmd_control                  8 <>
41 MRPD
50 L2CPD Filter
60 Mirror Task.8.1.18.f.80.0.0.4                 27 <>
60 knl Ifstate                                    6 <>
60 KNL
70 MGMT.local                                    28 <>
70 MGMT_Listen./var/run/l2cpd_mgmt               26 <Accept>
70 SNMP Subagent./var/run/snmpd_agentx           25 <>
```

So far, so good; the L2 control protocol daemon is running on both REs, which is not the case when NSB is disabled. The differences between NSR and NSB operation become apparent when the functional result of the `l2cpd`, namely xSTP and LLDP, are compared between master and BU. In the MX implementation of NSB, you *don't* expect to see the same STP/LLDP state on both REs. As mentioned previously, for L2 only the master state is replicated and the BU does not actually run copies of the various protocol state machines; hence, for example, STP shows all interfaces as disabled on the current BU. As always, we start at the master:

```
{master}
jnpr@R1-RE0>show lldp

LLDP                   : Enabled
Advertisement interval : 30 seconds
Transmit delay         : 2 seconds
Hold timer             : 4 seconds
Notification interval  : 0 Second(s)
Config Trap Interval   : 0 seconds
Connection Hold timer  : 300 seconds

Interface      LLDP
all            Enabled

{master}
jnpr@R1-RE0>show lldp neighbors
Local Interface Chassis Id        Port info     System Name
xe-2/0/0        00:1f:12:b7:df:c0  xe-2/0/0     R2-RE0
xe-2/0/1        00:1f:12:b7:df:c0  xe-2/0/1     R2-RE0

{master}
jnpr@R1-RE0>show spanning-tree interface vlan-id 100

Spanning tree interface parameters for VLAN 100

Interface    Port ID    Designated      Designated         Port    State  Role
                         port ID        bridge ID          Cost
ae0            128:483      128:483   4196.001f12b88fd0      1000  FWD    DESG
ae1            128:484      128:484   4196.001f12b88fd0      1000  FWD    DESG
ae2            128:485      128:485   4196.001f12b88fd0      1000  FWD    DESG
```

And now on the BU, where no LLDP neighbors are shown and the VSTP state is found to differ from that found on the master:

```
{backup}
jnpr@R1-RE1>show lldp

LLDP                   : Enabled
Advertisement interval : 30 seconds
Transmit delay         : 2 seconds
Hold timer             : 4 seconds
Notification interval  : 0 Second(s)
Config Trap Interval   : 0 seconds
Connection Hold timer  : 300 seconds

Interface      LLDP
all            Enabled

{backup}
jnpr@R1-RE1>show lldp neighbors

{backup}
jnpr@R1-RE1>show spanning-tree interface vlan-id 100

Spanning tree interface parameters for instance 100

Interface    Port ID    Designated      Designated         Port    State  Role
                         port ID        bridge ID          Cost
ae0            128:483      128:483   4196.000000000000         0  DIS    DIS
ae1            128:484      128:484   4196.000000000000         0  DIS    DIS
ae2            128:485      128:485   4196.000000000000         0  DIS    DIS
```

While the state does differ, the lack of error message when executing these commands on the BU RE also validates that NSB is in effect. Expect an error when NSB is not enabled as the l2cpd process is not running on the BU RE. Here, only GRES is configured and the BU RE returns the expected errors:

```
{backup}[edit]
jnpr@R1-RE1# run show lldp
error: the l2cpd-service subsystem is not running

{backup}[edit]
jnpr@R1-RE1# run show spanning-tree interface vlan-id 100
error: the l2cpd-service subsystem is not running

{backup}[edit]
jnpr@R1-RE1#

{backup}[edit]
jnpr@R1-RE1# run show system processes | match l2cpd
```

It bears repeating that in the case of NSB on MX, the STP Finite State Machine (FSM) is not run on the BU, and therefore we expect to see the port states/role as disabled, as shown previously. Recall that at switchover, the replicated state is used to bootstrap the xSTP process on the new master and the result is hitless from the perspective of STP speakers (i.e., there is no topology change flag seen or generated as a result of a GRES event).

One Junos, Many Platforms, Multiple Behaviors

This book focuses on the MX. Most of the commands and concepts apply to EX switches as they also run Junos. While there is one Junos, there are many flavors based on the specific platform, As an example, the 11.4 HA documentation for EX switches indicates that when verifying NSB, the BU RE should have matching STP state, which is not the case for MX routing running 11.4 Junos, as shown in this chapter:

Verifying Nonstop Bridging on EX Series Switches

. . .

<previous steps omitted for brevity>

3. Log in to the backup RE.

4. Enter the same show command that you entered in Step 2 to collect the same information about the NSB-supported Layer 2 protocol on the backup RE:

```
user@switch>show spanning-tree interface
Spanning tree interface parameters for instance 0
Interface Port ID Designated Designated Port State Role
port ID bridge ID Cost
ge-0/0/0.0 128:513 128:513 8192.0019e2500340 1000 FWD DESG
ge-0/0/2.0 128:515 128:515 8192.0019e2500340 1000 BLK DIS
ge-0/0/4.0 128:517 128:517 8192.0019e2500340 1000 FWD DESG
ge-0/0/23.0 128:536 128:536 8192.0019e2500340 1000 FWD DESG
. . .
```

In similar fashion, the operation of the periodic packet management daemon (ppmd) is confirmed. Once again, verification involves use of a hidden show ppm CLI command; once again, a case of hidden not due to any inherent danger to the router or its operator per-se, but because the operational details of the daemon are not publicly documented and such a display cannot, therefore, be officially supported:

```
{master}
jnpr@R1-RE0>show ppm ?
Possible completions:
  adjacencies          Show PPM adjacencies
  connections          Show PPM connections
  interfaces           Show PPM interface entries
  objects              Show PPM opaque objects
  rules                Show PPM interface entries
  transmissions        Show PPM transmission entries
{master}
```

First, the command output from the master:

```
{master}
jnpr@R1-RE0>show ppm connections detail
Protocol      Logical system ID  Adjacencies      Transmissions
BFD           All                2                2
ESMC          None               0                0
STP           None               0                5
ISIS          None               1                1
```

```
PIM           None              0                 0
PFE (fpc2)    573               7                 7
VRRP          None              1                 1
PFE (fpc1)    581               0                 0
ESIS          None              1                 1
LACP          None              6                 6
OAMD          None              0                 0
LFM           None              0                 0
CFM           None              0                 0

Connections: 13, Remote connections: 2
```

The connection output is useful on a few fronts. First, it helps confirm the various clients for which PPM-based hello generation is supported. Note the presence of BFD, LACP, STP, VRRP, IS-IS, and Ethernet OAM. The display also confirms that `ppmd` has two BFD connections/clients.

You may note the conspicuous absence of BGP and OSPF. This is because BGP hellos are always RE-based; after all, the lowest supported hold timer is nine seconds, making subsecond hello generation a nonissue. And besides, BGP keepalives are TCP-based, and processing the related state is a lot to ask of a PFE. At this time, OSPF hellos are also RE-based, but as with BGP, centralized handling (on the RE) of OSPF hellos are not an issue as the default timers on a LAN interface support a 40-second dead timer that means

Keep in mind that an OSPF hello or BGP keepalive is different than a BFD session that supports OSPF or BGP as a client, in which case BFD provides rapid fault detection while allowing the client protocols to use default timers that are long enouigh to permit RE-based hello generation.

A listing of the distributed, nondistrubited, and non-BFD clients in the 11.4 Junos release is provided.

Distributed PPMD clients:

BFD
STP
LFM
CFM
LACP
VRRP

Nondistributed PPMD clients:

OSPF2
OSPF3
ISIS
ESIS
LDP
OAMD

PIM

Non-PPMD clients:

BGP
RSVP

More details about the PPMD clients are seen with the `transmissions detail` switches:

```
{master}
jnpr@R1-RE0>show ppm transmissions detail

Destination: 10.3.255.2, Protocol: BFD, Transmission interval: 150

Destination: 10.8.0.1, Protocol: BFD, Transmission interval: 150
Distributed, Distribution handle: 3906, Distribution address: fpc2

Destination: N/A, Protocol: STP, Transmission interval: 2000

Destination: N/A, Protocol: STP, Transmission interval: 2000

Destination: N/A, Protocol: STP, Transmission interval: 2000

Destination: N/A, Protocol: STP, Transmission interval: 2000

Destination: N/A, Protocol: STP, Transmission interval: 2000

Destination: N/A, Protocol: ISIS, Transmission interval: 9000

Destination: N/A, Protocol: VRRP, Transmission interval: 1000

Destination: N/A, Protocol: ESIS, Transmission interval: 60000

Destination: N/A, Protocol: LACP, Transmission interval: 1000
Distributed, Distribution handle: 4001, Distribution address: fpc2

Destination: N/A, Protocol: LACP, Transmission interval: 1000
Distributed, Distribution handle: 4002, Distribution address: fpc2

Destination: N/A, Protocol: LACP, Transmission interval: 1000
Distributed, Distribution handle: 4017, Distribution address: fpc2

Destination: N/A, Protocol: LACP, Transmission interval: 1000
Distributed, Distribution handle: 4023, Distribution address: fpc2

Destination: N/A, Protocol: LACP, Transmission interval: 1000
Distributed, Distribution handle: 4027, Distribution address: fpc2

Destination: N/A, Protocol: LACP, Transmission interval: 1000
Distributed, Distribution handle: 4033, Distribution address: fpc2

Transmission entries: 16, Remote transmission entries:
```

The output confirms that the two BFD connections shown by `ppmd` are, in fact, the two used in this example to protect the IS-IS adjacency and the IBGP session between the two routers. This output is also quite useful for making it clear the former is distributed to FPC2 while the latter is not, meaning it is an RE-based BFD session. There's more detail on this a little later.

The final display shows the various clients that the ppmd process is aware of. In this example, most have their hello functions handled by the ppmd process itself, as confirmed in the previous command output for those connections shown as being distributed to a FPC. Those that are not distributed have their hellos handled by the RE, but the ppmd process is still aware of these sessions and monitors their state.

```
{master}
jnpr@R1-RE0>show ppm adjacencies
Protocol    Hold time (msec)
BFD         450
BFD         450
ISIS        27000
VRRP        3960
ESIS        180000
LACP        3000
LACP        3000
LACP        3000
LACP        3000
LACP        3000
LACP        3000
```

And now, the same commands executed on the BU:

```
{backup}
jnpr@R1-RE1>show ppm connections
Protocol       Logical system ID
BFD            All
ESMC           None
ISIS           None
PIM            None
STP            None

Connections: 5, Remote connections: 0

{backup}
jnpr@R1-RE1>show ppm adjacencies
Protocol    Hold time (msec)
BFD         450
BFD         450

Adjacencies: 2, Remote adjacencies: 0
```

These displays contain some real gold. The reader is encouraged to study the BFD and PPM-related output carefully while mulling back over all the NSR points made thus far; perhaps there is a landmine in addition to any gold mines, but again, more on that later.

Perform a NSR

Well, come on, what are you waiting for? Are you chicken? Everyone else is doing NSRs, and all indications to this point are that the system is ready. Here, the typical method of inducing a GRES event is used, shown with the `acquire` form executed on the current BU. The same result can be had on the current master with the `request chassis routing-engine master release no-confirm` form.

```
{backup}
jnpr@R1-RE1>request chassis routing-engine master acquire no-confirm
Resolving mastership...
Complete. The local routing engine becomes the master.
```

Initially, the NSR appears to have gone well, but they always do. Your heart sinks as you see traffic loss begin tallying on the router tester, and soon thereafter it's confirmed that the IBGP session has flapped on the new master:

```
{master}
jnpr@R1-RE1>show bgp summary
Groups: 2 Peers: 2 Down peers: 1
Table          Tot Paths  Act Paths Suppressed    History Damp State    Pending
inet.0               200        200          0          0          0          0
Peer                     AS      InPkt     OutPkt    OutQ   Flaps Last Up/Dwn State
  |#Active/Received/Accepted/Damped...
10.3.255.2      65000.65000       1012       1012       0       1           7 Active
192.168.0.1          65222        915       1021       0       0     7:37:10 100/100/100/0
  0/0/0/0

{master}
```

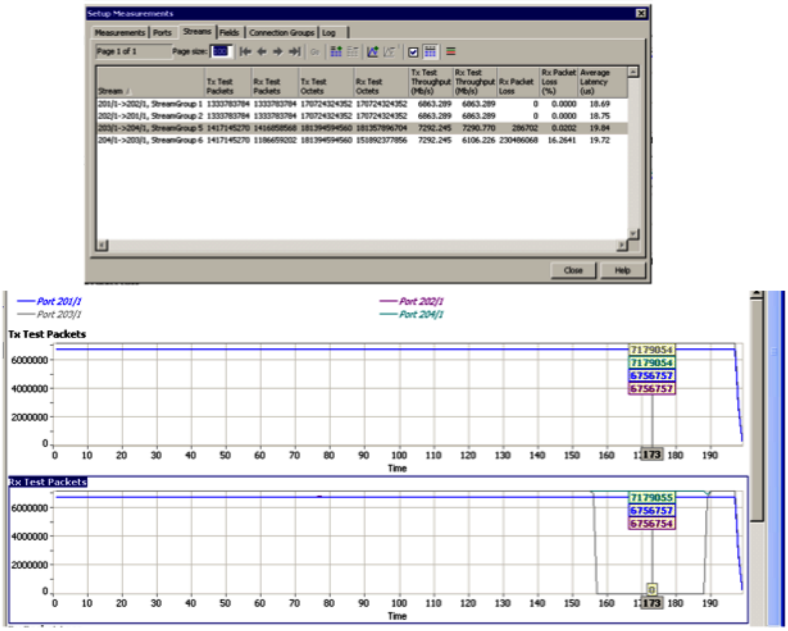
That was certainly not expected, at least not in terms of what was hoped to be a hitless event to both control and dataplane. Thinking quickly, you can confirm that IS-IS appears unaffected by the switchover, as should be the case for any NSR-supported protocol. Taking it where you can get it, this means you are batting 500; at least the NSR was not a complete disaster:

```
jnpr@R1-RE1>show isis adjacency
Interface             System         L State        Hold (secs) SNPA
ae0.1                 R2-RE0         2  Up                    21
```

Troubleshoot a NSR/NSB Problem. Though not shown, the IBGP flap and loss at NSR is consistently observed on subsequent NSRs, confirming it was not a one-of fluke. Oddly, the careful checks of replication state prior to the NSR would seem to indicate this is not an issue of unsupported NSR protocols; IS-IS and BGP have long been NSR supported, and so too has NSB for xSTP, distributed PPM, and GRES synchronization for all the other bits.

While the potential for a blatant NSR bug is always possible, given the rather vanilla nature of the configuration, and aforementioned indications of successful replication and general NSR readiness, it seems more likely this is somehow a configuration-related issue. In such cases, it's always good troubleshooting advice to reduce system complexity, or to eliminate areas that are known to be working so time is not wasted on the wrong protocol.

Figure 9-12 shows the nature of the loss observed during the NSR event.



Stream	Tx Test Packets	Rx Test Packets	Tx Test Octets	Rx Test Octets	Tx Test Throughput (Mb/s)	Rx Test Throughput (Mb/s)	Rx Packet Loss	Rx Packet Loss (%)	Average Latency (us)
201/1->202/1, StreamGroup 1	1333783784	1333783784	170724324352	170724324352	6863.289	6863.289	0	0.0000	18.69
202/1->201/1, StreamGroup 2	1333783784	1333783784	170724324352	170724324352	6863.289	6863.289	0	0.0000	18.75
203/1->204/1, StreamGroup 5	1417145270	1416858568	181394594560	181357896704	7292.245	7290.770	286702	0.0202	19.84
204/1->203/1, StreamGroup 6	1417145270	1186659202	181394594560	151892377856	7292.245	6106.226	230486068	16.2641	19.72

Figure 9-12. A Not-so-Hitless NSR.

The figure makes it clear the NSR was not hitless to the dataplane, but interestingly also exposes the fact that loss was limited to the Layer 3 portion of the network. Though not shown, previous testing with STP tracing on the switches confirmed that no TCN or other STP changes happened during a switchover, which also confirms that NSB is working as expected in this case. Combine this with the observation of 0 loss for Layer 2 traffic, and it seems you have already narrowed this down to a NSR as opposed to NSB issue, a fact that in itself is a major step in landing on the actual solution to this problem. Recall also that a previous display pointed to an issue with IBGP session, given that the EBGP sessions remained up through the NSR, as did the IS-IS adjacency.

While no specific protocol tracing was in effect, the BGP configuration did have the `log-updown` statement, and Junos is pretty good about logging things anyway. The syslog will have lots of entries after a GRES/NSR event. Junos has great pipe to match capability that makes it easy to find entries of interest. In this case, the following entries are found, shown here in chronological order:

```
Feb 15 20:34:30 R1-RE1 clear-log[10808]: logfile cleared
. .
```

```
Feb 15 20:34:40  R1-RE1 /kernel: mastership: sent other RE mastership loss signal
Feb 15 20:34:40  R1-RE1 /kernel: mastership: routing engine 1 becoming master
Feb 15 20:34:40  R1-RE1 /kernel: mastership: routing engine 1 became master
. . .
Feb 15 20:34:41  R1-RE1 rpd[10253]: RPD_BGP_NEIGHBOR_STATE_CHANGED: BGP peer 10.3.255.2
  (Internal AS 65000.65000) changed state from Established to Idle (event Restart)
. . .
Feb 15 20:34:41  R1-RE1 bfdd[10250]: BFDD_TRAP_MHOP_STATE_DOWN: local discriminator:
12,
  new state: down, peer addr: 10.3.255.2
```

The logs indicate the time of the GRES event and go on to confirm the idling of the IBGP connection to R2 lo0 address because of a restart event. It's quite the coincidence that the BFD session to the same lo0 address is also shown going down at the same time. Given both are shown to have occurred at the same time, it helps to recall that BFD's role is to rapidly detect failures and then notify its client, IBGP in this case, of the event to promote rapid reconvergence. As such, attention shifts to this being a BFD issue at NSR, rather than a BGP flap at NSR, which is, again, another significant step toward reaching the correct solution to a rather complex set of symptoms.

Given there were no log entries indicating interface/LACP flap during the NSR, and that the other BFD session that protects IS-IS (supported over the same link used to support the flapped IBGP session), remained up, the indication is this is not a link flap issue, which would point toward a possible GRES problem. It was also previously noted that the EBGP session did not flap, but it's not using BFD.

Factoring all this information shifts focus to what is so special about the IBGP BFD session. Then the answer strikes like a ton of bricks. Multihop BFD sessions are RE-based!

RE-based sessions cannot avail themselves of PFE-based uninterrupted packet generation through a GRES event, and therefore require a considerably longer hold time to be stable over a NSR. You recall that in previous PPM-related displays, only one of the BFD sessions was shown as distributed, which means the other must have been RE-based. Earlier in this chapter, at the end of the GRES section, there was a warning about this, stating that a minimum of 2,500 ms is recommended for RE-based sessions. You quickly look at the BGP configuration and confirm the issue:

```
{master}[edit]
jnpr@R1-RE1# show protocols bgp group int
type internal;
local-address 10.3.255.1;
family inet {
    unicast;
}
bfd-liveness-detection {
    minimum-interval 150;
    multiplier 3;
}
neighbor 10.3.255.2;
```

A quick change is made:

```
{master}[edit]
jnpr@R1-RE1# set protocols bgp group int bfd-liveness-detection minimum-interval 2500

{master}[edit]
jnpr@R1-RE1# commit
re1:
configuration check succeeds
re0:
commit complete
re1:
commit complete

{master}[edit]
jnpr@R1-RE1# run show bfd session address 10.3.255.2
                                                  Detect   Transmit
Address                  State     Interface      Time     Interval  Multiplier
10.3.255.2               Up                       7.500     2.500        3

1 sessions, 1 clients
Cumulative transmit rate 0.4 pps, cumulative receive rate 0.4 pps
```

With the change confirmed, the traffic generator is restarted in preparation for that hitless switchover you have been promised. Though not shown, the tester reports no loss, just as before. Meanwhile, the log is cleared on the soon-to-be-new master, and you begin monitoring the log in real time using the CLI's matching function to spot any changes to BGP or BFD.

```
{backup}
jnpr@R1-RE0>clear log messages

{backup}
jnpr@R1-RE0>monitor start messages | match "(bfd|bgp)"
```

And for the second time in a day, you have an opportunity to wield power that most humans cannot even begin to fathom. You pull the trigger on a NSR, this time switching control back to RE0. Drumroll, please:

```
{backup}
jnpr@R1-RE0>request chassis routing-engine master acquire no-confirm
Resolving mastership...
Complete. The local routing engine becomes the master.

{master}
. . .
```

And it's truly a case of "look to see what does not happen." That's part of the problem with Juniper's NSR. When it works, it works so well that customers have been known to make accusations that some form of trickery is at play; perhaps this misguided belief can account for the odd fetish some have developed regarding the yanking of a master RE from the chassis while emitting a brutish grunt?

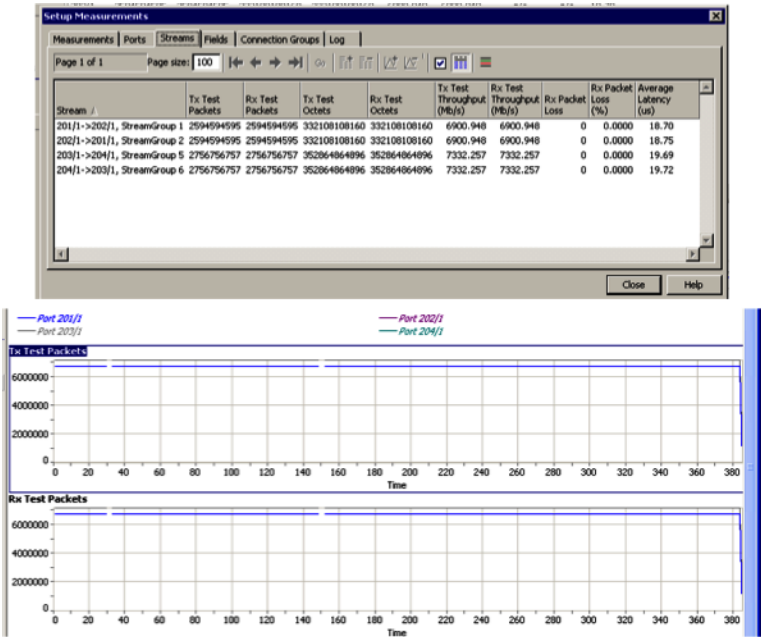
With the RE-based BFD session now running, the recommended long-duration timer all went to plan. No BFD, IS-IS, or BGP flap was detected on any peer or in the local syslog. All appears just as it did preswitchover on the old master:

```
jnpr@R1-RE0>show bgp summary
Groups: 2 Peers: 2 Down peers: 0
Table          Tot Paths  Act Paths Suppressed    History Damp State    Pending
inet.0               200        200          0          0          0          0
Peer                     AS      InPkt     OutPkt    OutQ   Flaps Last Up/Dwn
  State|#Active/Received/Accepted/Damped...
10.3.255.2      65000.65000         94         94       0       0       40:46
  100/100/100/0       0/0/0/0
192.168.0.1           65222         82         95       0       0       40:51
  100/100/100/0       0/0/0/0

{master}
jnpr@R1-RE0>show bfd session
                                                  Detect   Transmit
Address                  State     Interface      Time     Interval  Multiplier
10.3.255.2               Up                       7.500     2.500        3
10.8.0.1                 Up        ae0.1          0.450     0.150        3

2 sessions, 2 clients
Cumulative transmit rate 7.1 pps, cumulative receive rate 7.1 pps

{master}
jnpr@R1-RE0>show isis adjacency
Interface             System         L State        Hold (secs) SNPA
ae0.1                 R2-RE0         2  Up                   22

{master}
jnpr@R1-RE0>show spanning-tree interface vlan-id 100

Spanning tree interface parameters for VLAN 100

Interface    Port ID    Designated      Designated         Port    State  Role
                         port ID        bridge ID          Cost
ae0            128:483      128:483   4196.001f12b88fd0      1000  FWD    DESG
ae1            128:484      128:484   4196.001f12b88fd0      1000  FWD    DESG
ae2            128:485      128:485   4196.001f12b88fd0      1000  FWD    DESG
```

Sometime after the GRES event, the only syslog entries displayed are the expected reconnection of the **ppmd** process to the various FPCs. Part of the new master settling in to the new digs, as it were. This adds yet more proof that there was no BFD or BGP flap in this NSR event:

```
{master}
jnpr@R1-RE0>
*** messages ***
Feb 15 21:15:55  R1-RE0 fpc2 PPMAN: bfd conn ready
Feb 15 21:15:56  R1-RE0 fpc1 PPMAN: bfd conn ready
```

Figure 9-13 shows the traffic statistics for the final NSR.



Stream	Tx Test Packets	Rx Test Packets	Tx Test Octets	Rx Test Octets	Tx Test Throughput (Mb/s)	Rx Test Throughput (Mb/s)	Rx Packet Loss	Rx Packet Loss (%)	Average Latency (us)
201/1->202/1, StreamGroup 1	2594594595	2594594595	332108108160	332108108160	6900.948	6900.948	0	0.0000	18.70
202/1->201/1, StreamGroup 2	2594594595	2594594595	332108108160	332108108160	6900.948	6900.948	0	0.0000	18.75
203/1->204/1, StreamGroup 5	2756756757	2756756757	352864864896	352864864896	7332.257	7332.257	0	0.0000	19.69
204/1->203/1, StreamGroup 6	2756756757	2756756757	352864864896	352864864896	7332.257	7332.257	0	0.0000	19.72

Figure 9-13. The Hitless NSR, Truly a Thing of Beauty.

As advertised, there was absolutely zero, none, nada, zilch, zip loss, and that was for both bridges and routed traffic, as well as their associated protocols. That is the power of Junos-based NSR and NSB on the MX platform. A working NSR is truly a pleasure to behold.

NSR Summary

Given the grand success of the NSR case study, what more can be said? NSR is very cool and pretty darned impressive, even at this admittedly low scale. You can expect the same even with thousands of interfaces and protocol peers, as long as you practice the safe switchover guidelines covered in this section. There are many reasons for an NSR event to go less than gracefully. Bugs aside, most are documented, and most can be spotted before any switchover actually happens, if you know what to look for. If things do go bad, don't panic. NSR rarely makes things worse than if it were not enabled (granted, there can be NSR-specific bugs early on, but the technology has now had more than four years to mature), and if you keep a cool head and take some time to spot the lowest layer showing issues, you can usually quickly identify the protocol that is not behaving well and go from there.

Try to isolate issues as being GRES, GR, or NSR/NSB-related. Interface or LACP/BFD flaps tend to indicate issues with GRES- or centralized/RE-based sessions. In contrast, mismatched state on master versus backup tends to point to either an unsupported protocol or a replication error.

Remember to use NSR tracing when you feel replication is not completing or may be unstable, and be sure to make sure all replication tasks are complete before impressing your friends with your network's NSR Prowse. Premature back-to-back switchovers on a highly scaled system is a common reason for NSR to fail in random ways that can never be reproduced; given the high MTBF of Juniper REs, rapid or repeated switchovers are rarely expected in a production network.

NSR is the state of the art for network HA and is an enabling foundation for In-Service Software Upgrades (ISSU), the topic of the next section.

In-Service Software Upgrades

The Junos ISSU feature allows for a virtually hitless upgrade from one supported ISSU release to another. As mentioned previously, the foundation of ISSU is NSR, as during the ISSU process a GRES switchover occurs as the original standby becomes master to allow the old master (new backup) to be upgraded. Like GRES and NSR, ISSU can only be used on systems with redundant REs.

What is a supported ISSU release? In most cases, you are limited to no more than three major releases, and in theory any extended End-of-Life (EEOL) release to a current release should work. This section details current ISSU operation and restrictions for Trio-based MX routers running Junos 11.4.

ISSU Operation

At a high level, ISSU operation is pretty straightforward. Things begin when you include the `in-service-upgrade` switch rather than `add` when using the `request system software` command. The current master then pushes the new software bundle to the BU RE, where it's installed. After the BU RE reboots with the new software, the current master upgrades the PFE components in a sequential manner, which induces a brief dataplane outage known as a dark window. Next, a GRES event is triggered to make the old BU the new master, at which point the new BU (old master) begins installing the new software. When all goes to plan, you are left with both REs on the new software, now with RE1 as the master, all with no control plane hit and only a brief period of disruption to the data plane.

Figure 9-14 begins a sequence of figures that illustrate ISSU behavior on Trio-based MX routers.

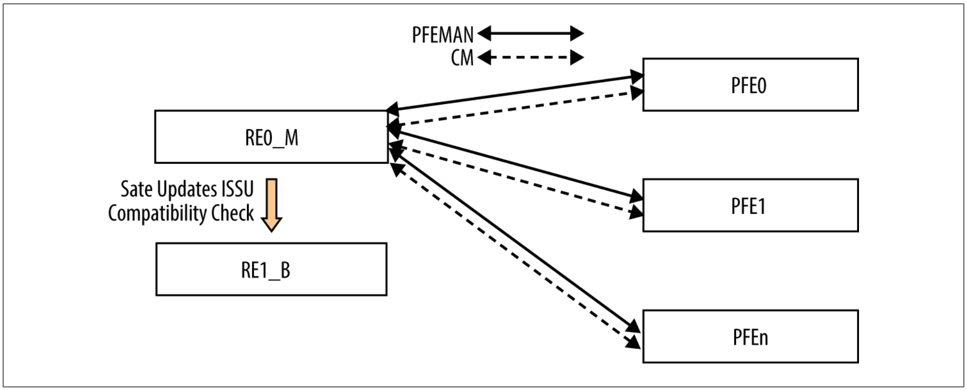


Figure 9-14. The ISSU Process.

Figure 9-14 shows the pre-ISSU state, with NSR and GRES in effect, and RE0 as the master (m) and RE1 as backup (b). Note that the master RE maintains two different types of chassis connections to each PFE in the system, namely a high-speed PFE manager link (`pfeman`) and a low-speed chassis management (CM) link. The CM thread supports the `chassisd` process and is used to bring up FPCs and to take PIC online/offline. The `pfeman` thread is responsible for handling IFD/IFL config messages and forwarding table updates from the RE.

Normal NSR and GRES replication and synchronization processes ensure that both REs have the same protocol and kernel state, and to begin both REs and the chassis are on the same version. At this point, the operator issues the `request system software in-service-upgrade <pkg>` command to initiate an ISSU, which begins a compatibility check to confirm the following:

This is an upgrade
That GRES, NSR, and NSB are in effect
Both REs are on the same version and that it's an ISSU supported release
The configuration is ISSU-supported
The hardware is ISSU-supported
No PPM or LACP processes are set to centralized

The ISSU process aborts if any are found to be false. In addition, the operator is warned when features or hardware are found that are known to result in control plane hits, for example when a PIC offline is required or when an unsupported BGP family is detected. In these cases, the operator is prompted to confirm if he or she wishes to proceed with an ISSU, as shown for the case of the `inet-mvpn` family in the 11.4 release:

```
. . .
Hardware Database regeneration succeeded
Validating against /config/juniper.conf.gz
mgd: commit complete
Validation succeeded
[edit protocols bgp group int family inet-mvpn]:
```

```
NSR is not supported; packet loss is expected
[edit protocols bgp group int family inet-mvpn]:
NSR is not supported; packet loss is expected
Do you want to continue with these actions being taken ? [yes,no] (no) no

error: ISSU Aborted!
Chassis ISSU Aborted
ISSU: IDLE

{master}
jnpr@R2-RE0>
```

Figure 9-15 shows the system state after the compatibility checks have passed and the new software bundle is pushed to RE1, which still functions in the BU role.

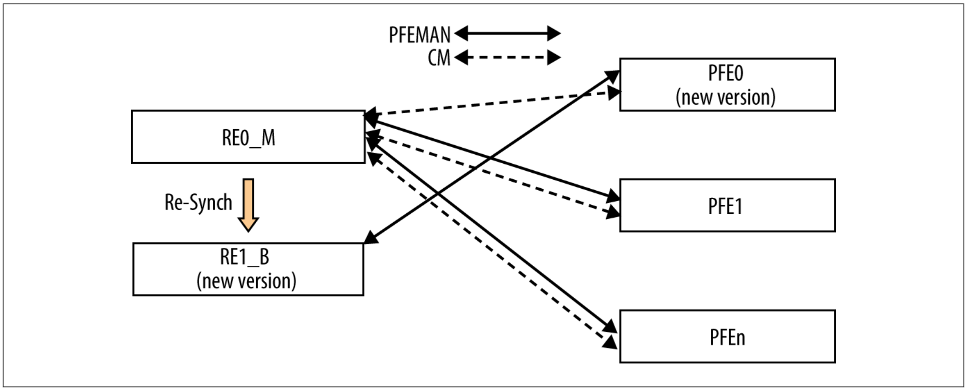


Figure 9-15. The ISSU Process Continued.

In Figure 9-15, RE1 has completed the reboot required to load its new software and is now on the new version, at which point it resynchronizes with the current master. After both REs are synchronized, the current master (RE0) begins upgrading the PFE components with the new software. On redundant systems, this is done in sequential fashion to minimize dataplane impact; otherwise, all components are upgraded in parallel. When the FPCs reboot, they too are now on the new code. Each upgraded PFE component now maintains a connection to each RE. The slow-speed CM connection is reformed to the current master (RE0) whereas the high-speed `pfeman` link is established to the current BU (RE1), which is running matched upgraded software. In Figure 9-15, only the first PFE has been loaded with the new software.

The next stage of the ISSU process is shown in Figure 9-16.

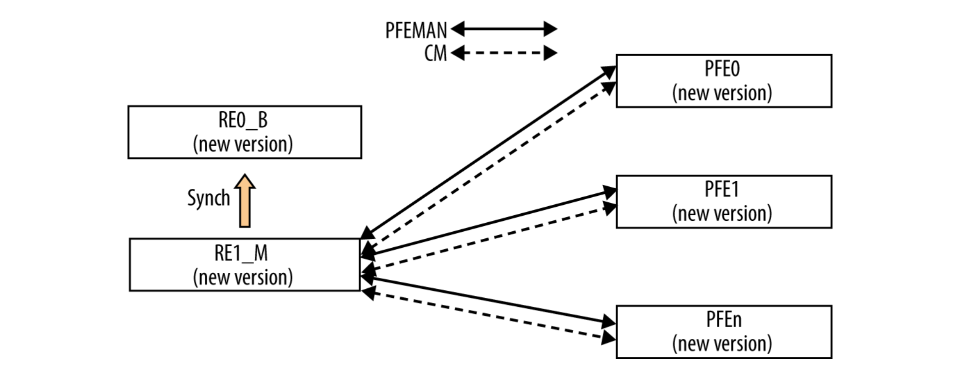


Figure 9-16. A Completed ISSU.

At this point, all PFE components are running the new software and a GRES event has occurred, placing RE1 into the master role. All FPCs reform their CM connections to the new master, and the old master, RE0, now in the BU role, has finished its installation and reboot, now on the same version as all other parts of the system. After booting, the new BU performs NSR/GRES replication and synchronization with RE1, the new master.

During this process, interface-specific and firewall filter statistics are preserved across an ISSU for Trio-based MX MPC/MIC interfaces; however, during the ISSU, counter and policer operations are disabled. The period of time the policers remain disabled is configuration dependant; note that all DDoS/host-bound policers remain in effect at all times. The pre-ISSU statistics are stored as binary large objects and then restored after the ISSU complete, a process that prevents statistics collection during the ISSU process.

ISSU Dark Windows

ISSU should be completely hitless to the control plane, but unlike a regular NSR, the need to push new code into the FPC forces two dark windows that are traffic-affecting:

- The first dark window affects only the host packet path from/to the RE and should last no longer than two seconds. This dark window is the result of the host to PFE path being temporarily unavailable while the PFE performs a reboot onto the new code.
- The second window impacts *both* the host packet and transit packet paths, and is expected to be no more than a few seconds per PFE; due to various Trio PFE optimizations, the actual dark windows can be far less. The second dark window results from the time necessary for the PFE to perform hardware synchronization with the new master RE. Trio PFEs have been optimized to reduce the duration of each dark window to an absolute minimum. Internal testing has shown that with one million routes, an I-chip PFE is expected to have up to seven seconds of dark

window while at this same scale a Trio-based PFE is expected to close its dark window within 700 ms, which is an order of magnitude improvement over the I-chip-based ADPCs.

- While short, the second window can disrupt Ethernet OAM sessions, which are rebuilt after the dark window. In contrast, the lack of dataplane hit in a conventional NSR event allows OAM session to remain operational through the switchover.

BFD and the Dark Window. Given that Bidirectional Forwarding Detection (BFD) is designed to rapidly detect forwarding plane problems, sessions with short timers can be disrupted during the ISSU dark window. This is prevented by temporarily increasing session detection and transmission timers during an ISSU event. After the upgrade, these timers revert to the values in use before the unified ISSU started.

Although it's not clear why you would want to, you can disable the BFD timer renegotiation feature by including the `no-issu-timer-negotiation` statement at the `[edit protocols bfd]` hierarchy level. When so configured, the BFD timers maintain their original values during the ISSU. You should expect BFD session flap and a resulting protocol and control plane disruption during the ISSU dark windows as a result. You can monitor the renegotiation behavior of BFD timers with ISSU tracing for BFD by including the `issu` statement at the `[edit protocols bfd traceoptions flag]` hierarchy.

> Use of the `no-issu-timer-negotiation` in conjunction with ISSU is not recommended. Depending on the detection intervals, some or all BFD sessions might flap during ISSU.

To demonstrate this adaptive BFD behavior, ISSU tracing is configured for BFD and an ISSU is started:

```
{master}[edit]
jnpr@R2-RE1# show protocols bfd
traceoptions {
    file bfd_issu_trace size 10m;
    flag issu;
}
```

The following BFD trace is observed on the new master shortly after the GRES:

```
{master}[edit]
jnpr@R2-RE1# run show log bfd_issu_trace

Feb 22 14:12:07 Check the Daemon ISSU state in the kernel
Feb 22 14:12:07 Daemon ISSU State <UNKNOWN>.
Feb 22 14:12:07 Chassisd ISSU State <IDLE>.
Feb 22 14:12:07 Handle mastership change
Feb 22 14:12:22 Sending the next session for ISSU timer negotiation 11
Feb 22 14:12:22 Revert session (discr 11) timers back to original values
Feb 22 14:12:22 Tx timer before is 20000000
Feb 22 14:12:22 Tx timer reverted back to 150000
Feb 22 14:12:22 Rx timer reverted back to 150000
Feb 22 14:12:22 Sending the next session for ISSU timer negotiation 13
Feb 22 14:12:22 Revert session (discr 13) timers back to original values
Feb 22 14:12:22 Tx timer before is 20000000
Feb 22 14:12:22 Tx timer reverted back to 2500000
Feb 22 14:12:22 Rx timer reverted back to 2500000
```

The trace confirms that both BFD sessions had their timers temporally increased to 20,000 ms (the trace output is in microseconds) to allow them to survive the ISSU dark windows, before being restored to their original values of 150 and 2,500 ms, respectively.

ISSU Layer 3 Protocol Support

Table 9-3 lists Layer 3 ISSU protocols support by release for Junos.

Trio-based MX routers do not support ISSU until release v11.2.

Table 9-3. Layer 3 ISSU Support by Release.

Protocol/Service	Minimum Junos Version
DHCP access model (subscriber access)	11.2 or later
IS-IS	9.0 or later
LDP	9.0 or later
LDP-based virtual private LAN service (VPLS)	9.3 or later
Layer 2 circuits	9.2 or later
Layer 3 VPNs using LDP	9.2 or later
Link Aggregation Control Protocol (LACP) on MX Routers	9.4 or later
OSPF/OSPFv3	9.0 or later
PPPoE access model (subscriber access)	11.4 or later
Protocol Independent Multicast (PIM)	9.3 or later
Routing Information Protocol (RIP) /RIPng	9.1 or later
Resource Reservation Protocol (RSVP) Ingress and Transit, L2/L3 VPN	10.2 or later

ISSU Layer 2 Support

Unified ISSU supports the Layer 2 Control Protocol process (`l2cpd`) on Trio-based MX routers. Recall that in Layer 2 bridge environment, spanning tree protocols (STP) share information about port roles, bridge IDs, and root path costs between bridges using special data frames called Bridge Protocol Data Units (BPDUs). The transmission of BPDUs is controlled by the `l2cpd` process. Transmission of hello BPDUs is important in maintaining STP adjacencies on the peer systems. The transmission of periodic packets on behalf of the l2cpd process is carried out by periodic packet management (PPM), which, by default, is configured to run on the PFE so that BPDUs are transmitted on time, even when the l2cpd process control plane is unavailable, a capability that keeps the STP topology stable during unified ISSU.

ISSU combined with NSB support for the l2cpd process ensures that the new master RE is able to take control during an ISSU without any disruptions in the Layer 2 control plane.

MX MIC/MPC ISSU Support

As of 11.4, ISSU on Trio-based MX routers is supported for the Modular Port Concentrators (MPCs) and Modular Interface Cards (MICs) listed in Table 9-4.

Table 9-4. MX Trio MPC/MIC ISSU Support in 11.4.

Type	Type (MPC or MIC)	Model
30-Gigabit Ethernet	MPC	MX-MPC1-3D
30-Gigabit Ethernet Queuing	MPC	MX-MPC1-3D-Q
60-Gigabit Ethernet	MPC	MX-MPC2-3D
60-Gigabit Ethernet Queuing	MPC	MX-MPC2-3D-Q
10-Gigabit Ethernet with SFP+, 16 ports	MPC	MPC-3D-16XGE-SFPP
Gigabit Ethernet MIC with SFP, 20 ports	MIC	MIC-3D-20GE-SFP
10-Gigabit Ethernet MICs with XFP, 2 ports	MIC	MIC-3D-2XGE-XFP
10-Gigabit Ethernet MICs with XFP, 4 ports	MIC	MIC-3D-4XGE-XFP
Tri-Rate Copper Ethernet MIC, 40 ports	MIC	MIC-3D-40GE-TX

ISSU: A Double-Edged Knife

Customers often state ISSU is really great feature, when it works. And that is the *issue* (to use a pun), with ISSU. It can almost seem impossible to predict the level of hit that you will experience in your next ISSU. In ideal situations, an ISSU completes with no control plane flap and only a small hit to the data plane known as a “dark window,” which occurs as new software is pushed into the PFE. The small hit that stems from the need to upgrade the PFE is why ISSU is said to be nearly hitless, as opposed to a basic NSR event, which as shown previously can be completely hitless.

The basic problem here is sheer complexity of the task at hand, which is akin to trying to change a car’s tires while it’s operating at high speed on an autobahn. The list of complicating factors includes:

Varying levels of protocol support by release and platform.
Varying levels of hardware (FPC/PIC) support by release and platform.

Relies on a successful GRES event with support that varies by release and platform.

Relies on a successful NSR event with support that varies by release and platform.

Even if bugs are found and fixed, the inherent nature of ISSU means that you will have to undergo a disruptive upgrade just to get on a version that contains a fix. Taking an upgrade hit just so that later you may upgrade nondisruptively makes little sense. Juniper does not currently provide hot-fix patch support for such upgrades. So, if there is a bug in your current code that affects ISSU, there is no hitless way out.

All of these are complicated as a function of the number of releases spanned by the ISSU. Performing an ISSU from and to the same version almost always works; going from an EEOL to a current release, well, maybe.

ISSU Restrictions

ISSU inherits all the caveats of GRES and NSR, and then adds a few of its own. As of the 11.4 Junos release, these restrictions include the following:

For Trio-based MX routers, v11.2 is the minimum supported ISSU release.

The ISSU procedure is not supported while upgrading from the 32-bit version of the Junos OS to the 64-bit version of the Junos OS. Currently, there is no hitless way to upgrade from a 32-bit to a 64-bit version of Junos.

ISSU only supports upgrades. There is currently no hitless downgrade method in Junos.

The master RE and backup RE must be running the same software version before you can perform an ISSU.

You cannot take any PICs online or offline during a unified ISSU.

On MX routers, ISSU does not support IEEE 802.1ag OAM and IEEE 802.3ah protocols. When an RE switchover occurs, the OAM hello times out during the dark window, which triggers a protocol convergence.

On MX routers, ISSU is not supported when clock synchronization is configured for Precision Time Protocol (PTP) and Synchronous Ethernet.

ISSU will abort if unsupported hardware, software, or protocols are found during the process.

PICs that are not ISSU-compatible (but are ISSU-supported) are brought offline during the ISSU process and then re-enabled at the end. The dark windows for these PICs can last several minutes.

In some cases, certain PIC combinations are not supported. This is because in a newer Junos version, a given feature or service may require additional PFE microcode memory and some configuration rules might limit certain combinations of PICs on particular platforms. If a PIC combination is not supported by the software version that the router is being upgraded from, the upgrade will be aborted. Likewise, if a PIC combination is not supported by the software version to which the router is being upgraded, the in-service software upgrade will abort, even if the PIC combination is supported by the software version from which the router is being upgraded.

Only the master instance supports ISSU. You can expect control and data plane disruption on nonmaster logical systems during an ISSU.

ISSU is not supported on MX80 routers, nor is it supported in an MX Series virtual chassis.

ISSU Troubleshooting Tips

ISSU has a lot going on underneath the covers. It's one of those things you start with a single command, and then a whole bunch of stuff happens in the background over a period of 20 to 30 minutes, and then it either works splendidly or it does not. In the latter case, the most common issue is some type of unexpected control or dataplane hit. For most of these cases, warnings are issued as part of the ISSU validation check, with the operator having to confirm a desire to proceed anyway, but such checks are never perfect. The result is that you should carefully heed all warnings issued, but the lack of warning in itself does not guarantee ISSU success.

In the other cases, the upgrade process may hang, in which case you can issue a `request system software abort in-service-upgrade` on the current master and look for any error messages that can lead you closer to the root cause of the malfunction. After an ISSU issue, the `show chassis in-service-upgrade` command on the new master to confirm the status of all FPCs and PICs. Some may have been taken offline during the process, and new microcode restrictions or other hard-to-predict interactions may keep them from coming back online.

Other tips and advice for ISSU include the following:

- You should perform a request system snapshot before starting an ISSU. That way if one of the REs fails to come back, you can reboot to alternate media to perform another snapshot and recover. Though somewhat rare and not necessarily related to ISSU, some software upgrades just fail.
- An ISSU is limited by overall success or failure or GRES and NSR. If you see the same type of failure in a GRES or NSR, then it's not an ISSU problem. Fix any NSR or GRES issue first.
- You should preverify ISSU compatibility of the software, hardware, and the configuration with the `request system software validate in-service-upgrade` command before scheduling a real ISSU event. There is no point in waiting until a maintenance window to then find out your system is not ISSU-compatible.
- As with NSR, it's best to test ISSU behavior under configurations that mimic production routers when you have to rely on a virtually hitless ISSU to maintain your network's SLA.
- You should have console access to both REs during an ISSU, and you should perform such actions during planned maintenance windows whenever possible, es-

pecially if you have not tested ISSU behavior in your network (see the previous point).

- Consider performing a `request system storage cleanup` on both REs before an ISSU. More disk space is never a bad thing when installing software and storing large binary objects.

ISSU Summary

ISSU is a great feature, really. But hitless or not, there is a lot of complexity at play. Complicate this with all the various software releases and that each network is a little different, and you quickly realize why it's just not feasible for Juniper to be able to test all possible ISSU permutations. There will always be the odd case of an ISSU blowing up, which is what people complain about on Internet forums, and this can cause folks to be shy of deploying ISSU. With an understanding of the ISSU feature, along with the underlying GRES/NSR technologies that it relies on, and a little homework, you can achieve virtually hitless software upgrades too.

> As with any complex software feature, it's best practice to test ISSU in a lab using test configurations that closely approximate those found in production routers to ensure you know what to expect when you perform an ISSU.

ISSU Lab

This section demonstrates several Junos HA features working together to support an ISSU. Namely GRES, NSR, and, of course, the ISSU feature itself. Figure 9-17 shows the modified lab topology for ISSU testing.

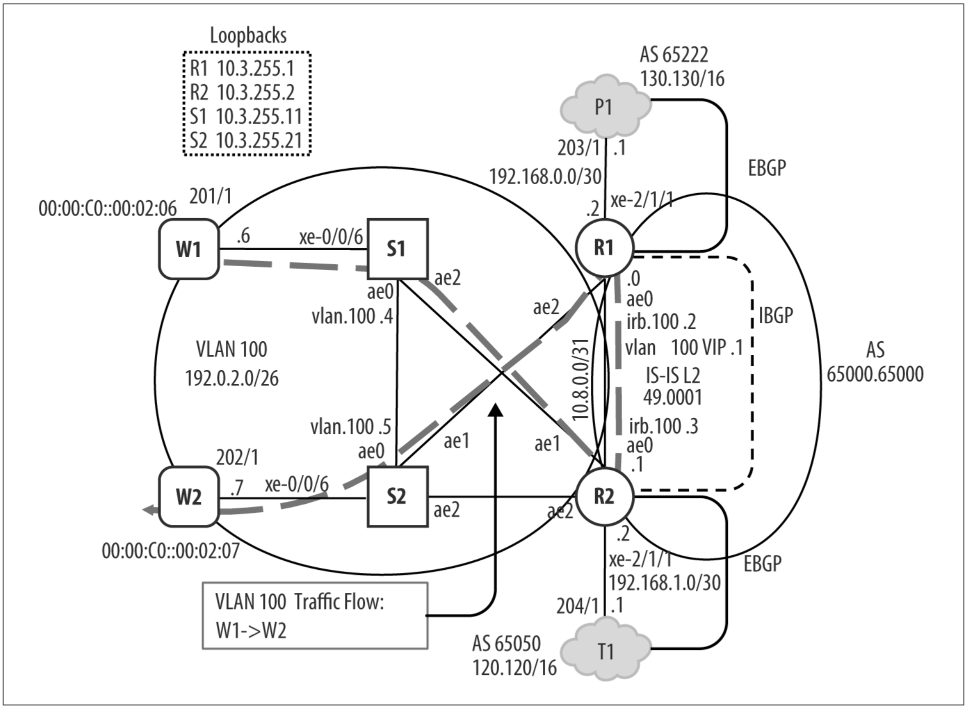


Figure 9-17. ISSU Test Topology.

As R1 has been getting all the attention lately, the plan has R2 being upgraded from its current 11.4R1.9 to 11.4R2.8. The primary modification, aside from the new Device under Test (DUT), is the deactivation of the ae1 interface at R1. This is done to force the Layer 2 traffic in VLAN 100 over the ae0 link, thereby placing R2 into the forwarding path for both the Layer 2 and Layer 3 traffic. The BGP traffic flows as in the previous NSR section, but now, VLAN 100 traffic arriving at S1's xe-0/0/6 interface is sent to R2 via its ae2 interface. Once there, the traffic is sent to R1 via the ae0 link, where it's then sent out R1's ae2 to reach S2 and the destination tester port; this convoluted forwarding path is shown on Figure 9-17 via the dashed line. Traffic sourced at S2 takes a similar path, going out its ae1 to R1, then out R1's ae0 over to R2, and then out R2's ae1 to reach the destination switch S1.

As noted previously, the result is that now both the Layer 2 and Layer 3 test traffic must transit the ae0 link between R1 and R2. The result is confirmed with a `monitor interface ae0` command at R1 while all four streams are flowing:

```
R2-RE0                        Seconds: 5                   Time: 16:17:03
                                                        Delay: 0/0/35
Interface: ae0, Enabled, Link is Up
Encapsulation: Flexible-Ethernet-Services, Speed: 20000mbps
                                                          Current delta
Traffic statistics:                                        [9869834252]
```

```
  Input bytes:             2703515009418 (13236474712 bps)     [9869753182]
  Output bytes:            3216196045130 (13236472096 bps)       [83130335]
  Input packets:             22597270063 (13935802 pps)          [83129598]
  Output packets:            27193182391 (13935798 pps)
Error statistics:                                                       [0]
  Input errors:                        0                                [0]
  Input drops:                         0                                [0]
  Input framing errors:                0                                [0]
  Carrier transitions:                 0                                [0]
  Output errors:                       0                                [0]
  Output drops:                        0
```

The output confirms the bidirectional symmetry of the flows and that both test streams are aggregated over the ae0 link, netting a combined rate of approximately 1.39 MPPS.

Verify ISSU Readiness

The DUT must be GRES and NSR-ready before ISSU can succeed. Refer back to the sections on GRES and NSR as needed for details on how those features work, and how to know when the DUT is in steady state after completing the requisite synchronization and replication.

If the GRES and NSR prerequisites are in place, your next step in ISSU validation is to run the `validate in-service-upgrade` command to determine if there is any software-, hardware-, or configuration-related issues that will prevent the DUT from performing a successful ISSU. As an example, consider the following output:

```
{master}
jnpr@R2-RE0>request system software validate in-service-upgrade jinstall-11.4R2.8-
  domestic-signed.tgz
Feb 18 16:27:38
Fetching package...
Checking compatibility with configuration
Initializing...
Using jbase-11.4R1.9
Verified manifest signed by PackageProduction_11_4_0
Verified jbase-11.4R1.9 signed by PackageProduction_11_4_0
Using /var/home/jnpr/jinstall-11.4R2.8-domestic-signed.tgz

Verified jinstall-11.4R2.8-domestic.tgz signed by PackageProduction_11_4_0
Using jinstall-11.4R2.8-domestic.tgz
Using jbundle-11.4R2.8-domestic.tgz
Checking jbundle requirements on /
Using jbase-11.4R2.8.tgz
Verified manifest signed by PackageProduction_11_4_0
Verified jbase-11.4R2.8 signed by PackageProduction_11_4_0
Using /var/validate/chroot/tmp/jbundle/jboot-11.4R2.8.tgz
Using jcrypto-11.4R2.8.tgz
Verified manifest signed by PackageProduction_11_4_0
Verified jcrypto-11.4R2.8 signed by PackageProduction_11_4_0
Using jdocs-11.4R2.8.tgz
Verified manifest signed by PackageProduction_11_4_0
Verified jdocs-11.4R2.8 signed by PackageProduction_11_4_0
```

```
Using jkernel-11.4R2.8.tgz
Verified manifest signed by PackageProduction_11_4_0
Verified jkernel-11.4R2.8 signed by PackageProduction_11_4_0
Using jpfe-11.4R2.8.tgz
WARNING: jpfe-11.4R2.8.tgz: not a signed package
WARNING: jpfe-common-11.4R2.8.tgz: not a signed package
Verified jpfe-common-11.4R2.8 signed by PackageProduction_11_4_0
WARNING: jpfe-X960-11.4R2.8.tgz: not a signed package
Verified jpfe-X960-11.4R2.8 signed by PackageProduction_11_4_0
Using jroute-11.4R2.8.tgz
Verified manifest signed by PackageProduction_11_4_0
Verified jroute-11.4R2.8 signed by PackageProduction_11_4_0
Using jruntime-11.4R2.8.tgz
Verified manifest signed by PackageProduction_11_4_0
Verified jruntime-11.4R2.8 signed by PackageProduction_11_4_0
Using jservices-11.4R2.8.tgz
Using jservices-crypto-11.4R2.8.tgz
Hardware Database regeneration succeeded
Validating against /config/juniper.conf.gz
mgd: commit complete
Validation succeeded
[edit protocols bgp group int family inet-mvpn]:
NSR is not supported; packet loss is expected
[edit protocols bgp group int family inet-mvpn]:
NSR is not supported; packet loss is expected
```

In this example, the presence of an unsupported (but NSR-compatible) feature is detected, namely the `inet-mvpn` MP-BGP family. Here, the user is given a warning to expect packet loss as the related BGP sessions are expected to flap at GRES/NSR. As of the 11.4R1 release, the `inet-mvpn` family is considered NSR-compatible, which is not the same as NSR-supported, as the latter implies hitless operation. In contrast, an incompatible feature either prevents you from committing the requisite NSR configuration or results in an abort of the ISSU process when detected later as part of the ISSU validation checks. In this example, the `inet-mvpn` family is deemed to be unnecessary so the fix is simple; remove it from the configuration (flapping the related BGP sessions, by the way). Afterwards, the validate process is performed again:

```
{master}
jnpr@R2-RE0>request system software validate in-service-upgrade jinstall-11.4R2.8-
  domestic-signed.tgz
Feb 18 16:54:05
Fetching package...
Checking compatibility with configuration
Initializing...
Using jbase-11.4R1.9
Verified manifest signed by PackageProduction_11_4_0
Verified jbase-11.4R1.9 signed by PackageProduction_11_4_0
Using /var/home/jnpr/jinstall-11.4R2.8-domestic-signed.tgz
. . .
Using jservices-crypto-11.4R2.8.tgz
Hardware Database regeneration succeeded
Validating against /config/juniper.conf.gz
```

```
mgd: commit complete
Validation succeeded
```

The output is truncated to save space, but this time it's clear the process completes with no warnings or failures. With confirmation of GRES and NSR readiness (shown in previous sections), and no stumbling blocks toward an ISSU from the current 11.4R1.9 to the planned 11.4R2.8 release, it's time to try the actual ISSU.

Perform an ISSU

In this section, you perform ISSU to upgrade an MX router from 11.4R1.9 to 11.4R2.8 with minimal disruption.

Before the ISSU is performed, the router tester is restarted, zero traffic loss is confirmed for all streams, all EBGP sessions are up, and the DUT reports GRES and NSR readiness via the `show system switchover` and `show task replication` commands on the backup and master, respectively. Though not shown, the desired Junos software package (`jinstall`) is copied to R2's RE0, the current master, using FTP or SCP. Some time stamps are added to the following ISSU to give a sense of the time scale involved in an ISSU in the 11.4 release.

To begin, the starting version is confirmed at R2:

```
{master}
jnpr@R2-RE0>show version
Feb 18 16:57:26
Hostname: R2-RE0
Model: mx240
JUNOS Base OS boot [c]
JUNOS Base OS Software Suite [11.4R1.9]
JUNOS Kernel Software Suite [11.4R1.9]
JUNOS Crypto Software Suite [11.4R1.9]
JUNOS Packet Forwarding Engine Support (M/T Common) [11.4R1.9]
JUNOS Packet Forwarding Engine Support (MX Common) [11.4R1.9
. . .
```

The `in-service-upgrade` command takes a few options:

```
{Master }
jnpr@R2-RE0>request system software in-service-upgrade jinstall-11.4R2.8-
  domestic-signed.tgz ?
Possible completions:
<[Enter]>            Execute this command
  no-copy              Don't save copies of package files
  no-old-master-upgrade  Don't upgrade the old master after switchover
  reboot               Reboot system after adding package
  unlink               Remove the package after successful installation
  |                    Pipe through a command
{Master}
jnpr@R2-RE0> request system software in-service-upgrade jinstall-11.4R2.8-
  domestic-signed.tgz
```

The most useful are the `reboot` and the `no-old-master-upgrade` switches. The former automatically reboots the new BU so it can complete installation of the new software. The default is for the new BU to wait until the operator instructs it to reboot. This leaves the new BU running on the old software image pending completion of the new software installation, which can only complete at reboot. The `no-old-master-upgrade` switch is used to only upgrade the current BU/new master. This allows you to perform an ISSU and test drive the new software without fully committing to it, in that if you find unanticipated issues in the new software, you can quickly recover with a NSR/GRES back to the old master, which is still on the old code. If you omit this switch and upgrade both REs (the default), only to later wish you had not, then your fastest recovery method is to reboot to alternate media to perform a new snapshot.

You did remember to snapshot with the old, stable version, prior to the ISSU, right?

And now the actual ISSU begins; in this example, the software package is in the user's home directory as opposed to being in `/var/tmp`, the latter being the preferred practice; we like to live on the edge here. This example shows the default form of the `in-service-upgrade` command, which is to say no optional switches are used.

```
{master}
jnpr@R2-RE0> request system software in-service-upgrade jinstall-11.4R2.8-
  domestic-signed.tgz
Feb 18 16:59:07
Chassis ISSU Check Done
ISSU: Validating Image
Checking compatibility with configuration
Initializing...
Using jbase-11.4R1.9
Verified manifest signed by PackageProduction_11_4_0
Verified jbase-11.4R1.9 signed by PackageProduction_11_4_0
Using /var/tmp/jinstall-11.4R2.8-domestic-signed.tgz
Verified jinstall-11.4R2.8-domestic.tgz signed by PackageProduction_11_4_0
Using jinstall-11.4R2.8-domestic.tgz
Using jbundle-11.4R2.8-domestic.tgz
Checking jbundle requirements on /
Using jbase-11.4R2.8.tgz
Verified manifest signed by PackageProduction_11_4_0
Verified jbase-11.4R2.8 signed by PackageProduction_11_4_0
Using /var/validate/chroot/tmp/jbundle/jboot-11.4R2.8.tgz
Using jcrypto-11.4R2.8.tgz
Verified manifest signed by PackageProduction_11_4_0
Verified jcrypto-11.4R2.8 signed by PackageProduction_11_4_0
Using jdocs-11.4R2.8.tgz
Verified manifest signed by PackageProduction_11_4_0
Verified jdocs-11.4R2.8 signed by PackageProduction_11_4_0
Using jkernel-11.4R2.8.tgz
Verified manifest signed by PackageProduction_11_4_0
Verified jkernel-11.4R2.8 signed by PackageProduction_11_4_0
Using jpfe-11.4R2.8.tgz
WARNING: jpfe-11.4R2.8.tgz: not a signed package
WARNING: jpfe-common-11.4R2.8.tgz: not a signed package
Verified jpfe-common-11.4R2.8 signed by PackageProduction_11_4_0
```

```
WARNING: jpfe-X960-11.4R2.8.tgz: not a signed package
Verified jpfe-X960-11.4R2.8 signed by PackageProduction_11_4_0
Using jroute-11.4R2.8.tgz
Verified manifest signed by PackageProduction_11_4_0
Verified jroute-11.4R2.8 signed by PackageProduction_11_4_0
Using jruntime-11.4R2.8.tgz
Verified manifest signed by PackageProduction_11_4_0
Verified jruntime-11.4R2.8 signed by PackageProduction_11_4_0
Using jservices-11.4R2.8.tgz
Using jservices-crypto-11.4R2.8.tgz
Hardware Database regeneration succeeded
Validating against /config/juniper.conf.gz
mgd: commit complete
Validation succeeded
ISSU: Preparing Backup RE
Pushing bundle to re1
Installing package '/var/tmp/jinstall-11.4R2.8-domestic-signed.tgz' ...
Verified jinstall-11.4R2.8-domestic.tgz signed by PackageProduction_11_4_0
Adding jinstall...
Verified manifest signed by PackageProduction_11_4_0

WARNING:     This package will load JUNOS 11.4R2.8 software.
WARNING:     It will save JUNOS configuration files, and SSH keys
WARNING:     (if configured), but erase all other files and information
WARNING:     stored on this machine.  It will attempt to preserve dumps
WARNING:     and log files, but this can not be guaranteed.  This is the
WARNING:     pre-installation stage and all the software is loaded when
WARNING:     you reboot the system.

Saving the config files ...
NOTICE: uncommitted changes have been saved in /var/db/config/juniper.conf.pre-install
Installing the bootstrap installer ...

WARNING:     A REBOOT IS REQUIRED TO LOAD THIS SOFTWARE CORRECTLY. Use the
WARNING:     'request system reboot' command when software installation is
WARNING:     complete. To abort the installation, do not reboot your system,
WARNING:     instead use the 'request system software delete jinstall'
WARNING:     command as soon as this operation completes.

Saving package file in /var/sw/pkg/jinstall-11.4R2.8-domestic-signed.tgz ...
Saving state for rollback ...
Backup upgrade done
Rebooting Backup RE
```

At this stage, all validation checks have completed and a copy of the software package is pushed to the current backup for installation. Part of the installation process of any jinstall is a reboot:

```
Rebooting re1
ISSU: Backup RE Prepare Done
Waiting for Backup RE reboot
```

Meanwhile, the console connection to the BU RE shows that it begins its reboot approximately 20 minutes after the ISSU process started:

```
{backup}
jnpr@R2-RE0> request system reboot
Reboot the system ? [yes,no] (no) yes


*** FINAL System shutdown message from jnpr@R2-RE0 ***

System going down IMMEDIATELY
. . . .

Feb 18 17:19:33
Shutdown NOW!
. . .
```

Back at the master, things proceed with an indication that the BU RE has rebooted and that GRES synchronization has completed; note this synchronization is between RE0 on the old version and RE1 on the new version:

```
GRES operational
Initiating Chassis In-Service-Upgrade
Chassis ISSU Started
ISSU: Preparing Daemons
ISSU: Daemons Ready for ISSU
ISSU: Starting Upgrade for FRUs
ISSU: Preparing for Switchover
ISSU: Ready for Switchover
Checking In-Service-Upgrade status
  Item           Status                  Reason
  FPC 1          Online (ISSU)
  FPC 2          Online (ISSU)
Resolving mastership...
Complete. The other routing engine becomes the master.
ISSU: RE switchover Done
. . .
```

As this stage, the current master has upgraded the PFE components, completing the second dark window, and has performed the GRES so that RE1, now running the new software, becomes master. Back on RE1's console, we see a proof of the new version and a timestamp as to when the GRES occurred:

```
--- JUNOS 11.4R2.8 built 2012-02-16 22:46:01 UTC
{backup}
regress@R2-RE1> set cli timestamp
Feb 18 17:16:16
CLI timestamp set to: %b %d %T
```

The GRES occurs.

```
{master}
regress@R2-RE1> show system uptime
Feb 18 17:17:40
Current time: 2012-02-18 17:17:40 PST
System booted: 2012-02-18 17:12:17 PST (00:05:23 ago)
Protocols started: 2012-02-18 17:13:22 PST (00:04:18 ago)
Last configured: 2012-02-18 17:13:51 PST (00:03:49 ago) by root
 5:17PM  up 5 mins, 1 user, load averages: 0.23, 0.26, 0.15
```

Back at the RE0, which you recall is now the new BU, we see the local software installation begin. In this example, the `request system software` command did not include the `reboot` option, so the new BU waits for the reboot command to finish its installation:

```
ISSU: Upgrading Old Master RE
Installing package '/var/tmp/jinstall-11.4R2.8-domestic-signed.tgz' ...
Verified jinstall-11.4R2.8-domestic.tgz signed by PackageProduction_11_4_0
Adding jinstall...
Verified manifest signed by PackageProduction_11_4_0

WARNING:     This package will load JUNOS 11.4R2.8 software.
WARNING:     It will save JUNOS configuration files, and SSH keys
WARNING:     (if configured), but erase all other files and information
WARNING:     stored on this machine.  It will attempt to preserve dumps
WARNING:     and log files, but this can not be guaranteed.  This is the
WARNING:     pre-installation stage and all the software is loaded when
WARNING:     you reboot the system.

Saving the config files ...
NOTICE: uncommitted changes have been saved in /var/db/config/juniper.conf.pre-install
Installing the bootstrap installer ...

WARNING:     A REBOOT IS REQUIRED TO LOAD THIS SOFTWARE CORRECTLY. Use the
WARNING:     'request system reboot' command when software installation is
WARNING:     complete. To abort the installation, do not reboot your system,
WARNING:     instead use the 'request system software delete jinstall'
WARNING:     command as soon as this operation completes.

Saving package file in /var/sw/pkg/jinstall-11.4R2.8-domestic-signed.tgz ...
Saving state for rollback ...
ISSU: Old Master Upgrade Done
ISSU: IDLE
```

At this state, ISSU has completed its work and so enters the idle state. The operator completes the upgrade of RE0 with a `reboot` command. As noted previously, the `reboot` switch could have been added to automate this stage of the upgrade, but either way, RE0 is ready to reboot at approximately 17:19:33. Based on the numbers, this means that in the JMX lab it took about 20 minutes for ISSU to complete, as the process started at approximately 16:59, but note you still need to reboot RE0 for software installation to complete, which adds another 5 to 10 minutes or so.

```
{backup}
jnpr@R2-RE0>request system reboot
Reboot the system ? [yes,no] (no) yes


*** FINAL System shutdown message from jnpr@R2-RE0 ***

System going down IMMEDIATELY


Feb 18 17:19:33
Shutdown NOW!
Reboot consistency check bypassed - jinstall 11.4R2.8 will complete installation
```

```
  upon reboot
[pid 37449]

{backup}
jnpr@R2-RE0>
```

Sometime later, RE0 is confirmed to boot to the new version, where it continues to function as BU:

```
. . .
Database Initialization Utility
RDM Embedded 7 [04-Aug-2006] http://www.birdstep.com
Copyright (c) 1992-2006 Birdstep Technology, Inc.  All Rights Reserved.

/var/pdb/profile_db initialized

Profile database initialized
Local package initialization:.
kern.securelevel: -1 -> 1
starting local daemons:
. . .

--- JUNOS 11.4R2.8 built 2012-02-16 22:46:01 UTC
{backup}
regress@R2-RE0>

{backup}
regress@R2-RE0> show system uptime
Current time: 2012-02-18 17:28:32 PST
System booted: 2012-02-18 17:26:41 PST (00:01:51 ago)
Protocols started: 2012-02-18 17:27:47 PST (00:00:45 ago)
Last configured: 2012-02-18 17:27:57 PST (00:00:35 ago) by root
 5:28PM  up 2 mins, 1 user, load averages: 0.54, 0.29, 0.12
```

The uptime at RE0 is used to confirm how long it took to complete its software installation and begin functioning as a BU RE. It shows that it started protocol processing at 17:27:47, indicating that the total ISSU process, to include rebooting the new BU, took about 27 minutes.

Confirm ISSU

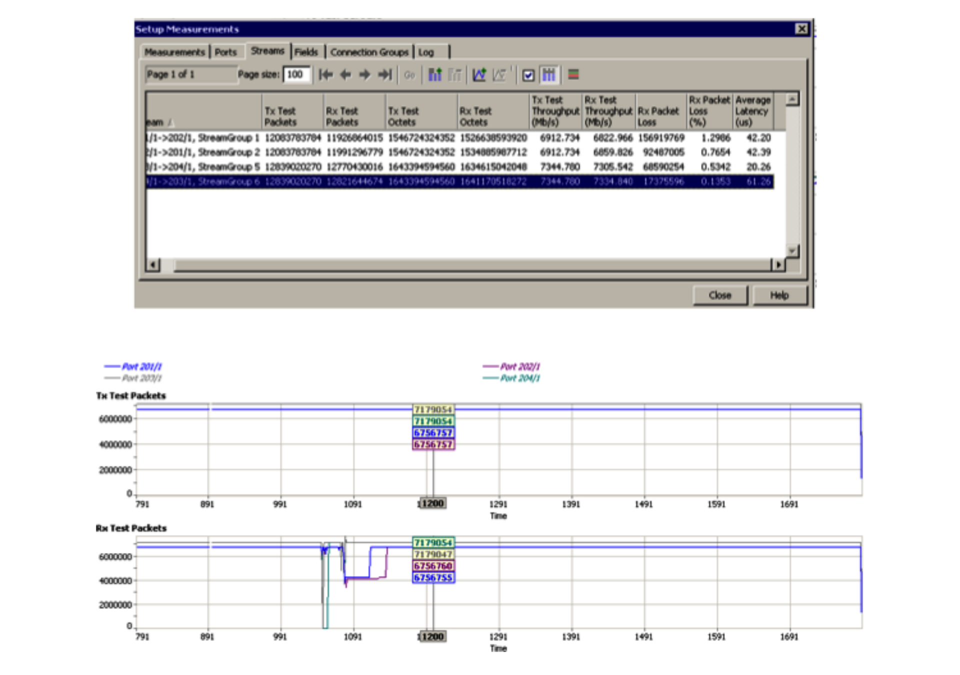
The operator is generally among the first to know if an ISSU was “virtually hitless” or not. In this example, the `show chassis in-service-upgrade` command is executed on the new master, where it reports that all FPCs are online post-ISSU, as expected given R2’s hardware and configuration is ISSU supported.

```
{master}
regress@R2-RE1>show chassis in-service-upgrade
Feb 18 17:18:58
  Item           Status                  Reason
  FPC 1          Online
  FPC 2          Online
```

In addition, the syslog on the new master is searched for any signs of control plane flap. It's a good idea to look for any BFD flaps if problems are seen in the control plane, as BFD is generally the first to go down when things go bad:

```
{master}
regress@R2-RE1>show log messages | match bfd

Feb 18 17:17:38  R2-RE1 fpc2 PPMAN: bfd conn ready
Feb 18 17:17:41  R2-RE1 fpc1 PPMAN: bfd conn ready
Feb 18 17:17:49  R2-RE1 bfdd[1445]: LIBJSNMP_NS_LOG_INFO:
  INFO: ns_subagent_open_session: NET-SNMP version 5.3.1 AgentX subagent connected

{master}
regress@R2-RE1>show log messages | match bgp
```

The log does not report any BFD or BGP flap, which jives nicely with the attached router tester's view, which in this example shows no BGP connection flap. The control plane is expected to be stable for NSR/ISSU-supported protocols, so this part checks out. A small dataplane hit is expected, however, so attention shifts to traffic loss through the issue and the resulting GRES. Figure 9-18 shows traffic statistics for the ISSU experiment.



eam	Tx Test Packets	Rx Test Packets	Tx Test Octets	Rx Test Octets	Tx Test Throughput (Mb/s)	Rx Test Throughput (Mb/s)	Rx Packet Loss	Rx Packet Loss (%)	Average Latency (us)
1/1->202/1, StreamGroup 1	12083783784	11926864015	1546724324352	1526638593920	6912.734	6822.966	156919769	1.2986	42.20
2/1->201/1, StreamGroup 2	12083783784	11991296779	1546724324352	1534885987712	6912.734	6859.826	92487005	0.7654	42.39
3/1->204/1, StreamGroup 5	12839020270	12770430016	1643394594560	1634615042048	7344.780	7305.542	68590254	0.5342	20.26
4/1->203/1, StreamGroup 6	12839020270	12821644674	1643394594560	1641170518272	7344.780	7334.840	17375596	0.1353	61.26

Figure 9-18. ISSU Test Results.

As expected, the tester reports some loss in both the Layer 2 and Layer 3 test streams. Based on the packet rate and number of packets lost, the dark windows can be reverse engineered. In this example, there was a period of approximately five seconds during which there was some level of traffic loss during the course of the ISSU. The graph indicates that the hit was shorter for the Layer 3 traffic, which was impacted first, and a bit longer for the Layer 2 traffic. It should be stressed that in this lab, the AE0 and AE1 interfaces at R2 are served by different Trio PFEs. Given that each PFE is upgraded in sequence, the loss in this experiment represents the effects of two commutative dark window periods. Therefore, the loss for links that are housed solely within a single PFE is expected to be half the duration shown in Figure 9-18.

The lack of control plane flap coupled with a small window of loss is in keeping with the design principles of ISSU and confirms a successful in-service upgrade.

Summary

Junos offers numerous HA features that can significantly improve network reliability and resiliency. These features are almost mandatory if you want to achieve the much bandied about "five 9s" reliability. After all, to meet 99.999% uptime, you are only allowed 5.26 minutes of downtime per year (which works out to only 25.9 seconds/month or 6.05 seconds/day), and given it takes 20 minutes or longer just to upgrade a modern Juniper router, the need for ISSU becomes quite apparent.

GRES serves as the foundation for most HA features, as being able to switch to a BU RE without disrupting the PFE is critical to both the NSF and NSR/NSB building blocks. Routers that are equipped with a single RE cannot use GRES, or NSR, but you can always enable graceful restart to provide tolerance to control plane faults and thereby achieve nonstop forwarding. Those that want state-of-the-art reliability will have redundant REs in their routers, and such users can avail themselves of NSR and ISSU to get maximum reliability and uptime.

Chapter Review Questions

1. Which HA features work together?
   a. NSR
   b. NSB
   c. GRES
   d. GR helper mode
   e. All of the above
2. Which of the following is considered a negative test for NSR?
   a. Restart routing
   b. Request chassis RE master switch on master
   c. Request chassis RE master switch on backup
   d. Panic the kernel on master RE

3. Which is false regarding graceful restart?
    a. GR will abort if instability is detected
    b. GR is hitless to the control plane, but has a slight hit to the dataplane
    c. GR and NSR cannot be configured together
    d. GR requires protocol extensions and is not transparent to peer routers
4. Which is true regarding GRES?
    a. GRES and GR can be used together on machines with a single RE
    b. You must run the same Junos version on both REs
    c. You must synchronize configurations between REs
    d. You must wait at least 240 seconds between each GRES event
    e. Both b and c
5. What is true regarding NSR?
    a. You can use NSR in conjunction with graceful restart
    b. You can enable NSR on machines with a single RE to provide protection from rpd faults
    c. NSR replicates protocols messages which are independently processed on the BU RE
    d. NSR replicates the master RE's forwarding and routing state (FIB/RIB) to the backup RE
6. Which command tells you if the backup RE has completed GRES synchronization?
    a. Show task replication on master
    b. Show task replication on backup
    c. Show system switchover on master
    d. Show system switchover on backup
7. What command is used to confirm NSR replication?
    a. Show task replication on master
    b. Show task replication on backup
    c. Show system switchover on master
    d. Show system switchover on backup
8. What type of session is not handled by PPM when in the (default) distributed mode with regard to NSR and NSB?
    a. Single-hop BFD sessions
    b. LACP
    c. OSPF hellos
    d. BGP keepalives

9. What is the minimum setting for an RE-based BFD session to ensure it remains up through a NSR?
    a. 50 ms
    b. 150 ms
    c. 2,000 ms
    d. 2,500 ms
10. Which of the following BFD sessions are RE based on release 11.4?
    a. OSPF
    b. OSPF3
    c. IS-IS Level 1
    d. Single-hop EBGP sessions
    e. All of the above
11. Which are true regarding ISSU on the MX?
    a. Both REs must be on the same ISSU-supported release
    b. You must be running NSR and NSB
    c. You can expect two brief periods of traffic loss: one affecting RE sourced and the other impacting all traffic
    d. Some protocols or feature may not be supported, resulting in control and dataplane hits
    e. All of the above.
12. Which is true regarding BFD?
    a. The use of BFD and graceful restart together is not recommended
    b. BFD sessions are expected to flap during an ISSU due to the dark windows
    c. Multihop BFD sessions cannot be used with ISSU
    d. BFD is not supported for GRES because fast detection times cannot be maintained through a switchover
    e. All of the above

Chapter Review Answers

1. **Answer: A**. All work together. Only GR restart mode is not supported while NSR is in effect.
2. **Answer: A**. The JUNOS NSR implementation is not designed to be hitless through a local routing process restart. GR, in contrast, does work through such a restart, albeit with control plane disruption. The remaining methods are all viable NSR failover trigger methods.

3. **Answer: B**. When all goes to plan, there should be no dataplane hits in a graceful restart. This makes B the only false answer.
4. **Answer: D**. Only D is true. Matched versions and configurations are recommended with GRES and NSR, but are not technically mandated.
5. **Answer: C**. NSR is based on protocol replication with independent message processing on both REs. Using the same messages and processing rules, the backup RE computes a shadow RIB that is expected to math that on the master RE. The RIB/FIB state is not simply replicated between the two REs.
6. **Answer: D**. The `show system switchover` command is run only on the backup RE, which is where the `ksyncd` process runs and reports GRES synchronization status.
7. **Answer: A**. While various replication commands can be run on both REs, the `show task replication` command can only run on the master RE and is the best gauge of overall system NSR readiness.
8. **Answer: D**. BGP keepalives are always RE based. All other session types are handled in the PFE by the PPM process.
9. **Answer: D**. According to the 11.4 documentation, RE-based BFD sessions should use at least a 2,500 ms detection timer to ensure stability through a GRES event.
10. **Answer: B**. In the 11.4 release, IPv6 control protocols protected by BFD that use link local addressing (i.e., OSPF3 uses RE-based BFD session processing). Use at least a 2,500 ms detection timer for these sessions to ensure stability through an NSR- or ISSU-based GRES.
11. **Answer: E**. All options listed must be true to initiate an ISSU.
12. **Answer: A**. With suitably long detection times, multihop BFD sessions are supported for NSR, GRES, and ISSU. The temporary negotiation of longer BFD timers during ISSU prevents disruption due to dark windows. While there is nothing to prevent committing such a configuration, the simultaneous use of BFD and graceful restart is not a supported feature combination.

Index

We'd like to hear your suggestions for improving our indexes. Send email to *index@oreilly.com*.

Colophon

The animal on the cover of *Juniper MX Series* is the tawny-shouldered podargus (*Podargus humeralis*), a type of bird found throughout the Australian mainland, Tasmania, and southern New Guinea. These birds are often mistaken for owls and have yellow eyes and a wide beak topped with a tuft of bristly feathers. They make loud clacking sounds with their beaks and emit a reverberating, booming call.

These birds hunt at night and spend the day roosting on a dead log or tree branch close to the tree trunk. Their camouflage is excellent—staying very still and upright, they look just like part of the branch. The tawny-shouldered podargus is almost exclusively insectivorous, feeding rarely on frogs and other small prey. They catch their prey with their beaks rather than with their talons, and sometimes drop from their perch onto the prey on the ground. The bird's large eyes and excellent hearing aid in nocturnal hunting.

Tawny-shouldered podargus pairs stay together until one of the pair dies. After mating, the female lays two or three eggs onto a lining of green leaves in the nest. Both male and female take turns sitting on the eggs to incubate them until they hatch about 25 days later, and both parents help feed the chicks.

The cover image is from Wood’s *Animate Creation*. The cover font is Adobe ITC Garamond. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont’s TheSansMonoCondensed.