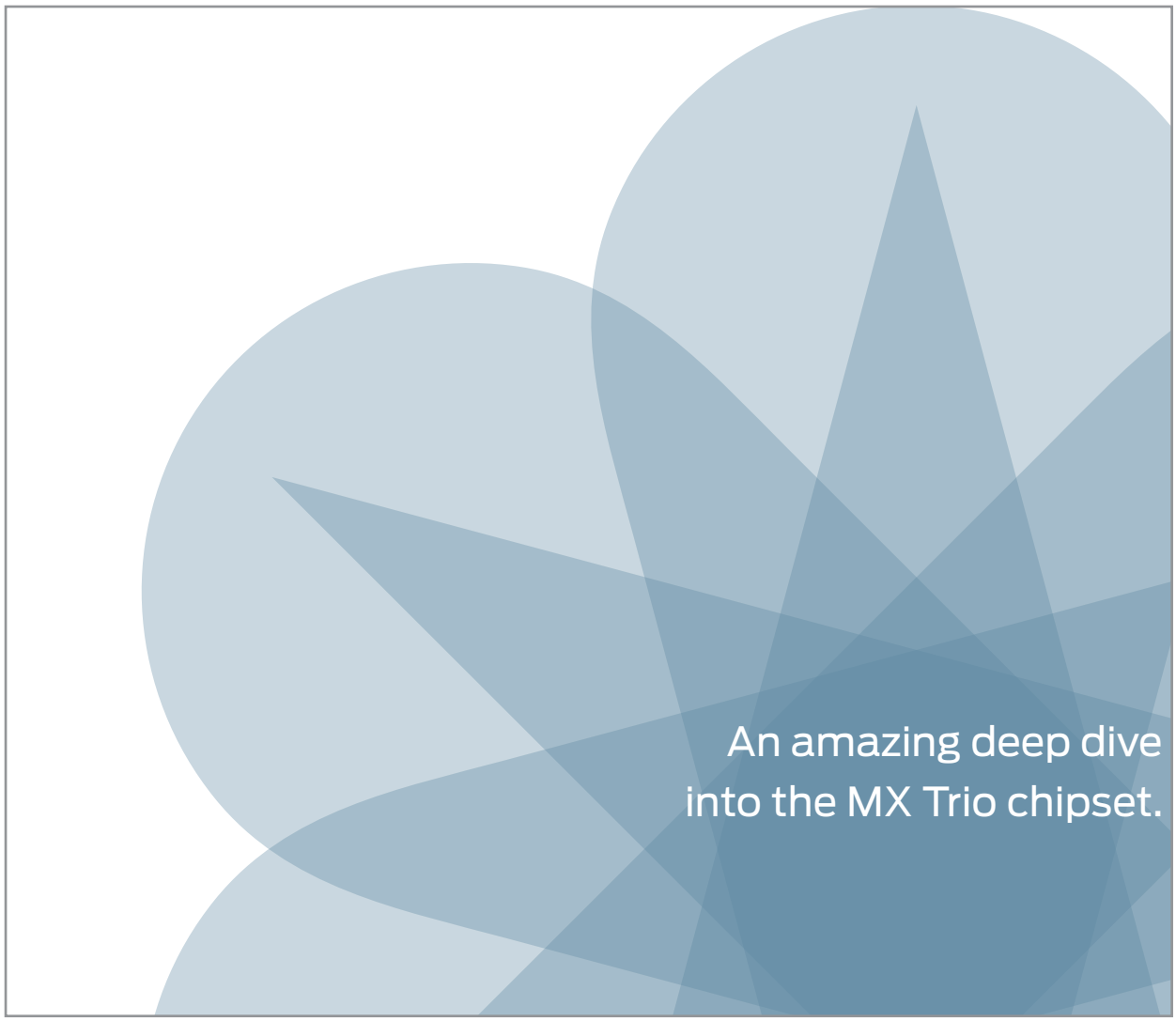
Junos® Networking Technologies

# THIS WEEK: AN EXPERT PACKET WALKTHROUGH ON THE MX SERIES 3D




By David Roy

# THIS WEEK: AN EXPERT PACKET WALKTHROUGH ON THE MX SERIES 3D

*This Week: An Expert Packet Walkthrough on the MX Series 3D* provides the curious engineer with a global view of the short life (a few milliseconds) of packets inside the Juniper Networks MX Series 3D routers. Though their life inside the router may be short, the packets are processed by an amazing ecosystem of next-generation technology.

Written by an independent network troubleshooting expert, this walkthrough is unlike any other. You'll learn advanced troubleshooting techniques, and how different traffic flows are managed, not to mention witnessing a Junos CLI performance that will have you texting yourself various show commands.

This book is a testament to one of the most powerful and versatile machines on the planet and the many engineers who created it. Sit back and enjoy a network engineering book as you travel inside the MX Series 3D.

*"This book is like a high-tech travel guide into the heart and soul of the MX Series 3D. David Roy is going where few people have gone and the troubleshooting discoveries he makes will amaze you. If you use the MX Series 3D, you have to read this book."*

*Kannan Kothandaraman, Juniper Networks Vice President*
*Product Line Management, Junos Software and MX Edge Routing*

## LEARN SOMETHING NEW ABOUT THE MX SERIES THIS WEEK:

- Understand the life of unicast, host, and multicast packets in the MX Series 3D hardware.
- Carry out advanced troubleshooting of the MX Series 3D Packet Forwarding Engines.
- Master control plane protection.
- Understand how Class of Service is implemented at the hardware level.

ISBN 978-1941441022


Published by Juniper Networks Books
www.juniper.net/books

JUNIPER NETWORKS

# This Week: An Expert Packet Walkthrough on the MX Series 3D

## By David Roy



JUNIPER NETWORKS

**Published by Juniper Networks Books**
Author: David Roy
Editor in Chief: Patrick Ames
Copyeditor and Proofer: Nancy Koerbel
Illustrations: David Roy
J-Net Community Manager: Julie Wider

ISBN: 978-1-941441-02-2 (print)
Printed in the USA by Vervante Corporation.

ISBN: 978-1-941441-03-9 (ebook)

Version History: v1, January 2015
2 3 4 5 6 7 8 9 10

This book is available in a variety of formats at:
http://www.juniper.net/dayone.

**About the Author**
David Roy lives and works in France where seven years ago he joined the Network Support Team of Orange France. He is currently responsible for supporting and deploying the IP/MPLS French Domestic Backbone. David holds a master's degree in computer science and started his career in a research and development team that worked on Digital Video Broadcasting over Satellite and then started working with IP technologies by designing IP solutions over Satellite for Globecast (an Orange Subsidiary). David is a Juniper Networks Expert holding three JNCIE certifications: SP #703, ENT #305, and SEC #144.

## Author's Acknowledgments

I would like to thank my wife, Magali, and my two sons, Noan and Timéo, for all their encouragement and support. A very special thank you to Antonio Sanchez-Monge from Juniper Networks for helping me during the project and who was also the main technical reviewer. A great thank you to Josef Buchsteiner, Steven Wong, and Richard Roberts for their deep technical review and interesting concept discussions. Finally I want to thank Qi-Zhong and Steve Wall from Juniper Networks, and Erwan Laot from Orange, for their incredibly useful feedback, and to Patrick Ames for his review and assistance.

*David Roy, IP/MPLS NOC Engineer, Orange France*
*JNCIE x3 (SP #703 ; ENT #305 ; SEC #144)*

## Technical Reviewers

- **Antonio "Ato" Sanchez Monge,** Network Architect - Telefonica (Advanced Services, Juniper Networks). I work with MX series (all the way from vMX to MX2020) in lab and production networks. My main activities are design, feature testing, and support.
- **Steven Wong**, Principal Escalation Engineer, Juniper Networks. I handle technical issues and help enhance the MX platform to provide a better user experience.
- **Josef Buchsteiner,** Principal Escalation Engineer, Juniper Networks. I resolve technical issues, drive diagnostics, and support capabilities on MX platforms.
- **Richard Roberts**, Network Architect - Orange (Professional Services, Juniper Networks). I work directly with David supporting him and his team since the introduction of the first generation of MX960 and now the latest MX2020 routers.
- **Qi-Zhong Cao**, Sr. Staff Engineer, Juniper Networks. My primary focus is DDOS protection of Juniper MX routers. I develop software components spanning the entire host path.
- **Steve Wall**, Test Engineer Sr. Staff, Juniper Networks. I do Product Delivery Testing for the MX and other platforms targeting their deployment into large service provider and Web 2.0 customer networks.
- **Babu Singarayan**, Sr. Staff Engineer, Juniper Networks. I work on MX-Trio architecture and development with expertise on MX forwarding and host-path.
- **Erwan Laot**, IP/MPLS NOC Engineer, Orange France. I've been working with David Roy and MX routers for several years, and both are equally resourceful and a pleasure to work with when addressing new technical challenges.
- **Michael Fort**, Sr. Staff Engineer, Juniper Networks. PFE/BRAS software architecture and development with a bias towards automation, performance, process, and forensics.

## Welcome to *This Week*

*This Week* books are an outgrowth of the extremely popular *Day One* book series published by Juniper Networks Books. *Day One* books focus on providing just the right amount of information that you can execute, or absorb, in a day. *This Week* books, on the other hand, explore networking technologies and practices that in a classroom setting might take several days to absorb or complete. Both libraries are available to readers in multiple formats:

- Download a free PDF edition at http://www.juniper.net/dayone.
- Get the ebook edition for iPhones and iPads at the iTunes Store>iBooks. Search for *Juniper Networks Books*.
- Get the ebook edition for any device that runs the Kindle app (Android, Kindle, iPad, PC, or Mac) by opening your device's Kindle app and going to the Kindle Store. Search for *Juniper Networks Books*.
- Purchase the paper edition at either Vervante Corporation (www.vervante.com) or Amazon (www.amazon.com) for prices between $12-$28 U.S., depending on page length.
- Note that Nook, iPad, and various Android apps can also view PDF files.

## What You Need to Know Before Reading

- You need to be *very* familiar with the Junos Operating System.
- You need to know basic Class of Service and multicast concepts for the second half of the book.

## After Reading This Book You'll Be Able To

- Understand the life of unicast, host, and multicast packets in MX Series 3D hardware
- Carry out advanced troubleshooting of the MX Series 3D hardware
- Master control plane protection
- Understand how class-of-service is implemented at hardware level

MORE? This book is not meant to replace MX Series 3D technical documentation that can be found at www.juniper.net/documentation, where there are key details, installation requirements, deployment guides, and network solutions.

## Author's Notes

The MX Series 3D Universal Edge Router is a mouthful. This book uses abbreviated terms such as *MX 3D*, and *MX Series 3D*, to focus on what's inside the device.

I have included notes and notation within device output and configurations. They are designated by several "less than" characters in succession followed by a boldface output font, such as shown here:

```
NPC0(R2 vty)# test xmchip 0 wo stats default 0 0 <<< third 0 means WAN Group 0
```

# Chapter 1

## An Extremely Quick MPC Overview

This book provides you with a global view of the short life (a few milliseconds) of packets inside Juniper Networks MX Series 3D routers. They have a short life inside the router, but as you will see, the packets are processed by many components.

As a network support engineer, I sometimes face issues that involve many vendors and their equipment in complex topologies. During these troubleshooting sessions, I found that one of the most interesting things about the Juniper routers is their rich troubleshooting toolbox.

The Junos OS is an awesome open platform – nothing is hidden and you can track the life of each packet inside the router whether it's a control or transit type of packet. I spent a lot of time reverse engineering how packets (control- and data plane-based) are handled and managed by the Juniper hardware and the Junos software. The result is this book, which focuses on the life of the packet inside the MX Series 3D hardware.

READY? If you are not familiar with the MX basic architecture, and if terms like MPC, MIC, PIC, SCB, PFE, etc., don't sound crystal clear to you, have a look at Chapter 3 of *This Week: A Packet Walkthrough on the M, MX, and T Series*, an introductory book that belongs in the *Day One* library: http://www.juniper.net/us/en/training/jnbooks/day-one/networking-technologies-series/a-packet-walkthrough/.

MX Series 3D includes components that are based on a common architecture and network philosophy. The MX 3D term was born with the very first release of the Trio chipset, and it has been kept throughout the evolution into newer chipset generations. This book focuses on the line cards that are based on the Trio architecture. All these line cards are called MPC (Modular PIC Concentrator).

More specifically, the lab for this book uses two MPC models: one 16x10GE MPC card and one MPC4e card. These particular line card models do not have MIC slots, so their PICs are integrated (built) into the MPC.

NOTE You can easily port most of the commands and concepts in this book to other MPC models—even the fabric-less MX models like MX5, MX10, MX80, or MX104, whose PFE is implemented in a single TFEB or Trio Forwarding Engine Board— if you need to. Last but not least, the Virtual MX or VMX will also be based on the Trio architecture.

## A Quick Overview Inside the MPC

Let's introduce the MPC's Packet Forwarding Engine (PFE) chips (or ASICs). Some of them are embedded on all types of MPCs while others are optional and are only available on certain MPC models. For more up-to-date information, please refer to the Juniper Networks website and technical documentation (http://www.juniper.net/documentation), which provides a complete feature and component list.

PFEs are made of several ASICs, which may be grouped into four categories:

- Routing ASICs: LU or XL Chips. LU stands for Lookup Unit and XL is a more powerful (X) version.
- Forwarding ASICs: MQ or XM Chips. MQ stands for Memory and Queuing, and XM is a more powerful (X) version.
- Enhanced Class of Service (CoS) ASICs: QX or XQ Chips. Again, XQ is a more powerful version.

- Interface Adaptation ASICs: IX (only on certain low-speed GE MICs) and XF (only on MPC3E).

NOTE Why two names for each Routing or Forwarding ASIC type? The names on the right correspond to more powerful components or second generation ASIC so they are implemented in newer PFE models.

The Enhanced CoS and Interface Adaptation ASICs are optional and are not included in all MPC/MIC models. Inversely, Routing and Forwarding ASICs are always present and constitute the core of an MPC's Packet Forwarding Engine (PFE).

As of the publication of this book there are three generations of 3D MPCs:

- First generation MPCs, containing the original Trio chipset with LU and MQ chips. This generation includes MPC types 1 and 2, as well as the MPC 16x10GE.
- Second generation MPCs, whose PFEs have LU and XM chips. This generation includes MPC3e and MPC4e.
- Third generation MPCs, whose PFEs have XL, XQ, and XM chips. This generation includes MPC5e and MPC6e.

The newer MPC5e and MPC6e (the MPC6e is for MX20*xx* routers) are beyond the scope of this book. So XL chips are not covered here. The chips fully covered in this book are the LU, MQ, and XM chips.

IMPORTANT Our selection of the features supported by MX 3D routers is tailored to the needs of this book.

One of the more useful Packet Forward Engine commands that you are going to use is the "jspec" shell command which allows you to know which ASICs are present in the MPC.

ALERT! *This is one of the most important paragraphs in this book!* Even though the author did not encounter any problems while executing the following PFE commands in lab scenarios, please remember that *shell commands are not supported in production environments*, and this book is no exception. In production networks, you should only execute these commands if you are instructed to do so by JTAC. Moreover, the shell commands and their output shown in this book are provided for the purpose of illustration only, and should not be taken as any kind of shell command guide.

Let's use the jspec command on the R2 MPC 16x10GE and MPC4e cards and see which ASICs are embeded on each card:

```
user@R2> request pfe execute command "show jspec client" target fpc11 | trim 5

 ID       Name
  1       LU chip[0]
  2       MQChip[0]
  3       LU chip[1]
  4       MQChip[1]
  5       LU chip[2]
  6       MQChip[2]
  7       LU chip[3]
  8       MQChip[3]
```

```
user@R2> request pfe execute command "show jspec client" target fpc0 | trim 5

 ID       Name
  1       LU chip[0]
  2       LU chip[4]
  3       XMChip[0]
  4       LU chip[1]
  5       LU chip[5]
  6       XMChip[1]
```

As you can see, both cards are made of MQ or XM chips and LU chips. Let's step back from the PFE internals for a bit, and see how the different MPC functional components (control plane microprocessor, PFE ASICs, etc.) are interconnected. There are several types of links:

- Ethernet: The linecard's CPU (a.k.a. µKernel's CPU) or control plane microprocessor "speaks" with the Routing Engine via two embedded Gigabit Ethernet interfaces (em0 and em1).
- PCIe: The linecard's CPU is in charge of programming the ASICs, pushing the forwarding information base (FIB) to the LU chip memory and the basic scheduling configuration to the MQ/XM chips. This CPU communicates with ASICs via a PCIe Bus.
- I2C: The I2C Bus allows the control components, hosted by the (S)CB, to monitor and retrieve environmental (power, temperature, status, etc.) information from the different MPC's components.
- HSL2: The PFE's ASICs communicate with each other and with the fabric chips via HSL2 (High Speed Link version 2) links. This is how the forwarding plane is actually implemented: every transit packet spends some time through HSL2 links.

### A Word on HSL2

High Speed Link Version 2 is a physical link technology that makes it possible to convey high speed data among ASICs in a same PFE but also between PFEs and the fabric. The data layer protocol over HSL2 allows channelization and supports error detection via a CRC mechanism. You can retrieve HSL2 links and their statistics by using the following microkernel shell command:

```
NPC0(R2 vty)# show hsl2 statistics
Cell Received (last)           CRC Errors (last)
-----------------------------------------------

LU chip(0) channel statistics :
LU chip(0)-chan-rx-0 <= XMChip(0)-chan-tx-135   526216395719869  (139077335)  0  (0)
LU chip(0)-chan-rx-1 <= XMChip(0)-chan-tx-134   526216395719869  (139077335)  0  (0)
[...]
```

TIP You can interact with the microkernel of a line card by launching the hidden (and *unsupported*) command `start shell pfe network fpc<slot>`. On fabric-less MXs, `fpc<slot>` is replaced with `tfeb0`.

### MPC Type 1

Let's start with the MPC type 1, which is a modular MPC that can host two MICs. Some MICs (like the 20x1GE MIC) host the specific ASIC called *IX*. IX manages the

physical interfaces and provides a first level of packet classification. MPC ype 1 only has one PFE, which is made of two ASICs or chips:

- The MQ chip is in charge of packet memory management, queuing packets, "cellification," and interfacing with the fabric planes. It features both fast on-chip (SRAM) memory and off-chip memory.
- The second ASIC is the LU chip. It performs packet lookup, packet firewalling and policing, packet classification, queue assignment, and packet modification (for example, push/swap/push MPLS headers, CoS remarks, etc.). This chip also relies on combined on-chip/off-chip memory integration. The FIB is stored in off-chip memory.

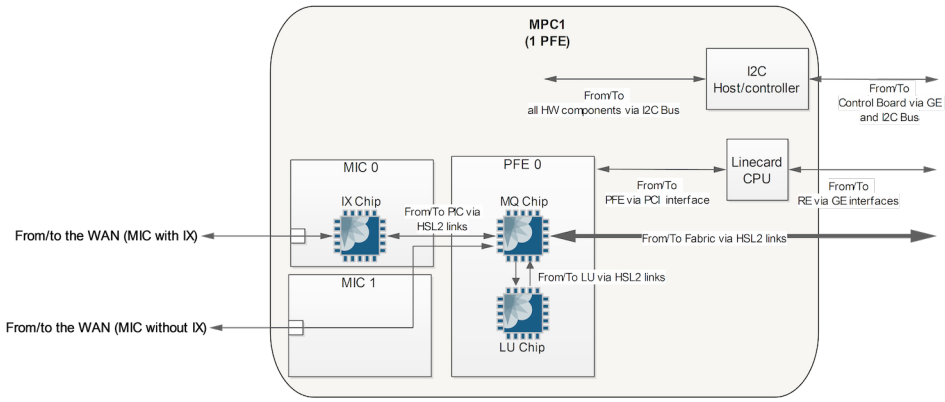



Figure 1.1 MPC1 Internal View

### MPC Type 2

The next MPC is the MPC Type 2. In this case, the diagram intentionally shows an enhanced queuing (EQ) MPC, to briefly present the QX Chip. QX provides rich queuing and advanced CoS features (like hierarchical per-VLAN queuing). MPC Type 2 has two PFEs (see "Repeat" notation in Figure 1.2) made of one MQ, one LU, and one QX chip each. Each PFE manages one MIC (with or without IX chip, depending on the model).

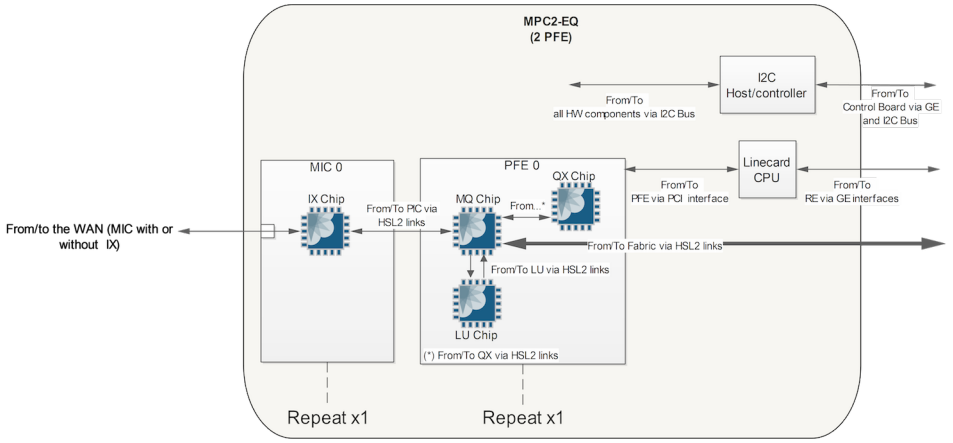



Figure 1.2 MPC2-EQ Internal View

### MPC 16x10GE

The famous monolithic MPC 16x10GE hosts four PFEs (4x10GE ports per PFE), each linked to an internal built-in PIC, as shown in Figure 1.3.

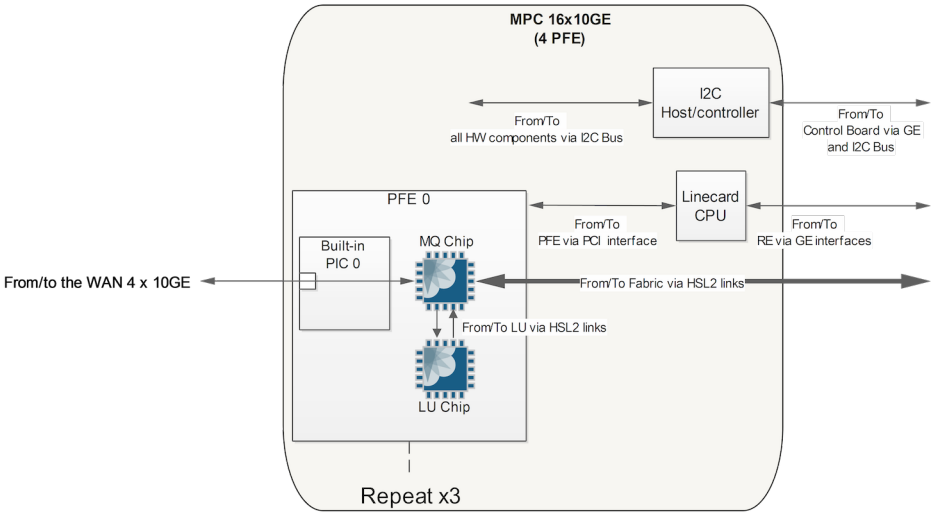



Figure 1.3 MPC 16x10GE Internal View

### MPC3e

The MPC3e, depicted in Figure 1.4, was the first MPC made with the next version of the MQ Chip called the *XM chip*. XM provides the same functionalities as MQ but with new features and scaling four times more than MQ. XM load balances the traffic towards four LU chips. The XM chip introduces the concept of *WAN Group* which can be considered as a virtual PFE. In case of the MPC3e, the WAN Group 0 manages the MIC 0 and WAN Group 1 manages the MIC 1. The MPC3e's XM chip doesn't directly connect to the fabric planes — actually, one XF ASIC, programmed in **`Fabric Offload mode`** plays the role of gateway between the PFE and the fabric.

NOTE This is the same XF ASIC that you'll find in SCBE cards, although on SCBEs XF is programed in *Standalone mode* instead. In the MPC3e, this chip just provides compatibility with legacy fabric chips of the original SCB model.

### MPC4e

The last, but not the least, MPC in this book is the MPC4e. MPC4e is a monolithic MPC and two models of MPC4e are available: the 32x10GE card and the 8x10GE+2x100GE card. MPC4e is made of two PFEs and you can see in Figure 1.5 that now XM itself is interconnected directly with the fabric planes. Moreover, you can again see the concept of WAN Group, but in the case of this monolithic card the WAN Group association is a little bit different than the MPC3e. Indeed. WAN Groups are assigned differently, depending on the WPC4e model:

- MPC4e 32x10GE: For PFE 0, WAN Group 0 is associated to the first 8x10GE ports and WAN Group 1 to the next eight ports. For PFE 1 this is the same for the remaining 16x10GE ports.
- MPC4e 8x10GE+2x100GE: For PFE 0, WAN Group 0 is associated to the first 4x10GE ports and WAN Group 1 to the first 1x100GE port. For PFE 1 this is the same for the remaining 4x10GE and 1x100GE ports.

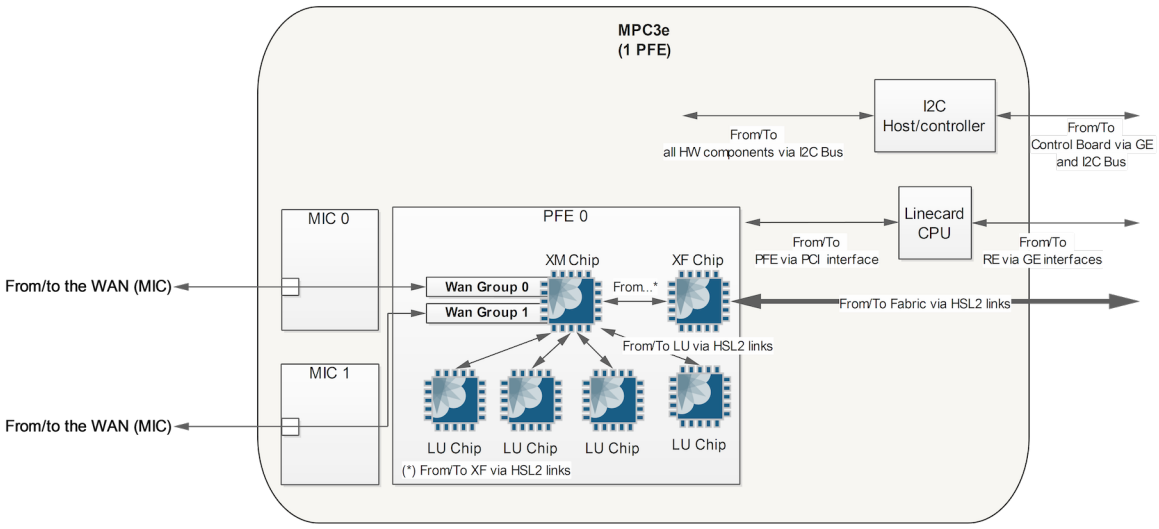



Figure 1.4 MPC3e Internal View

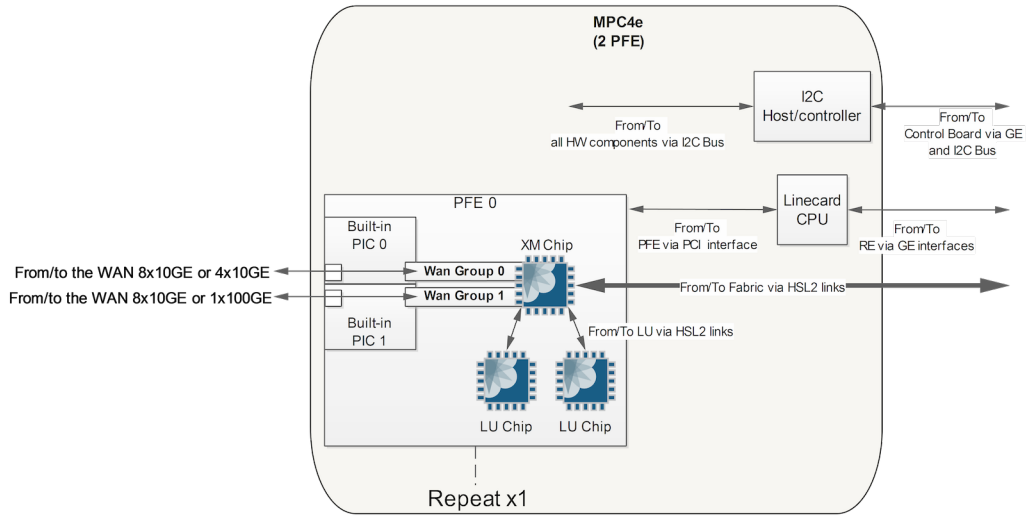



Figure 1.5 MPC4e Internal View

## PFE Numbering

This book refers to a PFE ID, which makes it sound like there is only one ID associated with a PFE, but actually, for any given PFE, there are two IDs assigned: the local ID, which has a meaning at the MPC level, and the global ID that uniquely identifies a PFE in a chassis.

The local ID depends on the type of MPC. Each MPC has a fixed number of PFEs between 1 and 4. The local PFE ID always starts at 0 and increments by 1 for the next PFE of the MPC. For example, the 16x10GE card has four PFEs numbered from 0 to 3, while the MPC4e has only two PFEs numbered from 0 to 1.

The global PFE ID is computed as follows:

```
GLOBAL_PFE_ID= 4 x MPC_SLOT_NUMBER + LOCAL_PFE_ID
```

The Junos OS assumes that each slot can have a maximum of four PFEs. If you refer to Figure 1.6 and focus on R2, the global PFE_IDs of the PFEs in MPC slot 0 are: 0 and 1 (for this MPC4e, global PFE_IDs 2 and 3 are dummy IDs). The global PFE_IDs of PFEs in MPC slot 11 are: 44, 45, 46, and 47 (the 16x10GE MPC has 4 PFEs).

## This Book's Topology

This book relies on a very simple topology made of five MX routers. Keep your eyes on R2 because it will be your *Device Under Test*. R2 is a MX960 with two MPCs, one 16x10GE card in slot 11, and one MPC4e (8x10GE+2x100GE) card in slot 0. The reason behind this choice is that MPC 16x10GE and MPC4e are representatives of the first and second generation of 3D chipsets, respectively. The R2 router uses SCBE fabric planes and it runs Junos 14.1R1.10. Figure 1.6 illustrates the physical topology used in this book's lab.

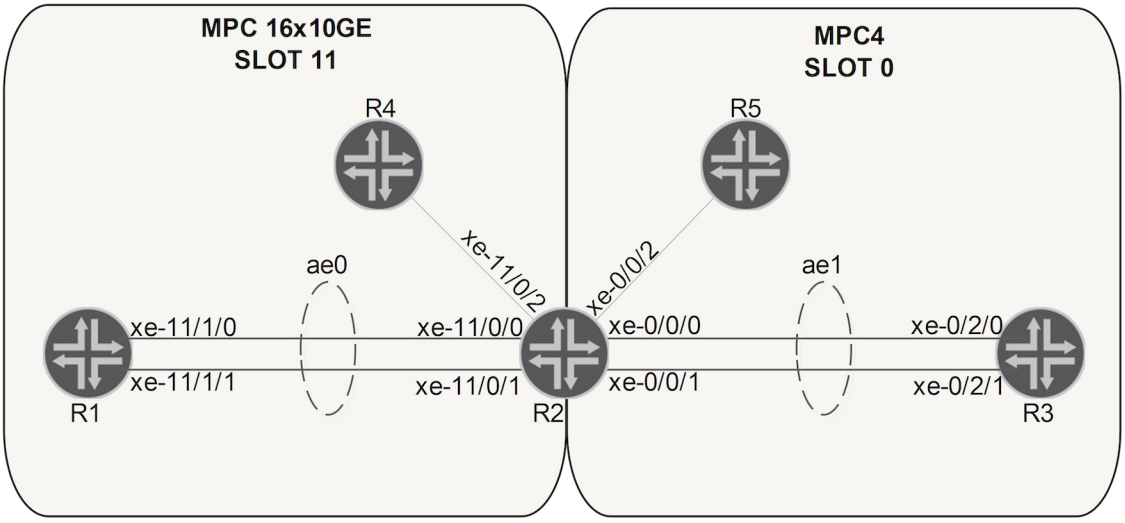



Figure 1.6 This Book's Lab Topology

## Summary

This was an extremely short history of MPCs. But it quickly reviews what you need to keep in mind in the lab and for the remainder of this book. For more specifics on each MPC see the Juniper technical documentation at http://www.juniper.net/documentation.

Now it's time to start following packets in the extraordinary MX Series 3D Universal Edge Router.

# Chapter 2

## Following a Unicast Packet

This chapter reviews in-depth concepts common to every kind of packet and MPC card. The first part concerns a *Unicast transit packet* and it will be especially detailed.

Other types of packets will be covered in subsequent chapters, but this chapter is where you first step into the water.

This chapter begins with basic functionalities like how a unicast transit packet is handled by the PFE, how it is forwarded, and how it is modified. Some specific subsections with more detailed concepts such as the case load balancing and CoS are included.

## Unicast Network Topology

Adding to the previous physical topology depicted in Chapter 1, Figure 2.1 illustrates more details regarding IP addresses and flows. IPv4 addresses are used but this entire analysis could also be applicable to IPv6 traffic as well. There is no dynamic protocol set in this topology – even LACP is disabled. Only static routes have been configured to ensure the routing of non-direct subnets.

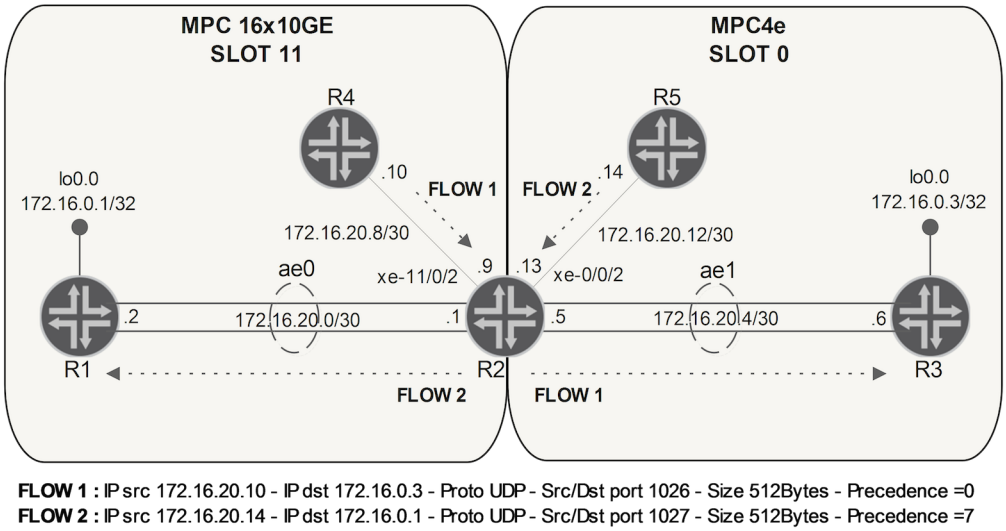



Figure 2.1 Unicast Transit Flows

You can refer back to Figure 1.6 for the port numbering.

IMPORTANT MX PFEs operate in packet mode. The word flow refers to a sequence of packets with identical properties. Don't think of it as a stateful flow.

The goal is to track the two UDP flows:

- Flow 1: A 1000 pps UDP stream from R4 to R3 loopback address in transit through R2 – with an IP precedence field = 0
- Flow 2: A 1000 pps UDP stream from R5 to R1 loopback address in transit through R2 – with an IP precedence field = 7

```
user@R2> show interfaces ae[0,1] | match "Phy|rate"
Physical interface: ae0, Enabled, Physical link is Up
```

```
  Input rate     : 0 bps (0 pps)
  Output rate    : 3953144 bps (1000 pps)
Physical interface: ae1, Enabled, Physical link is Up
  Input rate     : 0 bps (0 pps)
  Output rate    : 3953144 bps (1000 pps)
```

As you can see from the show command, R2's aggregated interfaces forward the 1000 pps of each flow.

Okay, it's time to dive into the Junos 3D hardware of R2 to better understand how these two flows received from R4 and R5 are forwarded to their destination. During this detailed analysis of the packet life, you will see many functional blocks implemented at several levels of the 3D linecards. For each functional block both CLI commands and PFE shell commands are provided to help you troubleshoot or better understand the analysis.

## Handling MAC Frames

Let's start to analyze the two flows arriving to R2 at the interface xe-11/0/2 for Flow 1 (MPC 16x10GE card) and the interface xe-0/0/2 for Flow 2 (MPC4e).

When a packet is received by the router, it is first handled by the MAC controller. This component provides an interface between the PHY layer and the MAC layer and delivers Ethernet Frames to the PFE. Figure 2.2 shows you the detailed view.

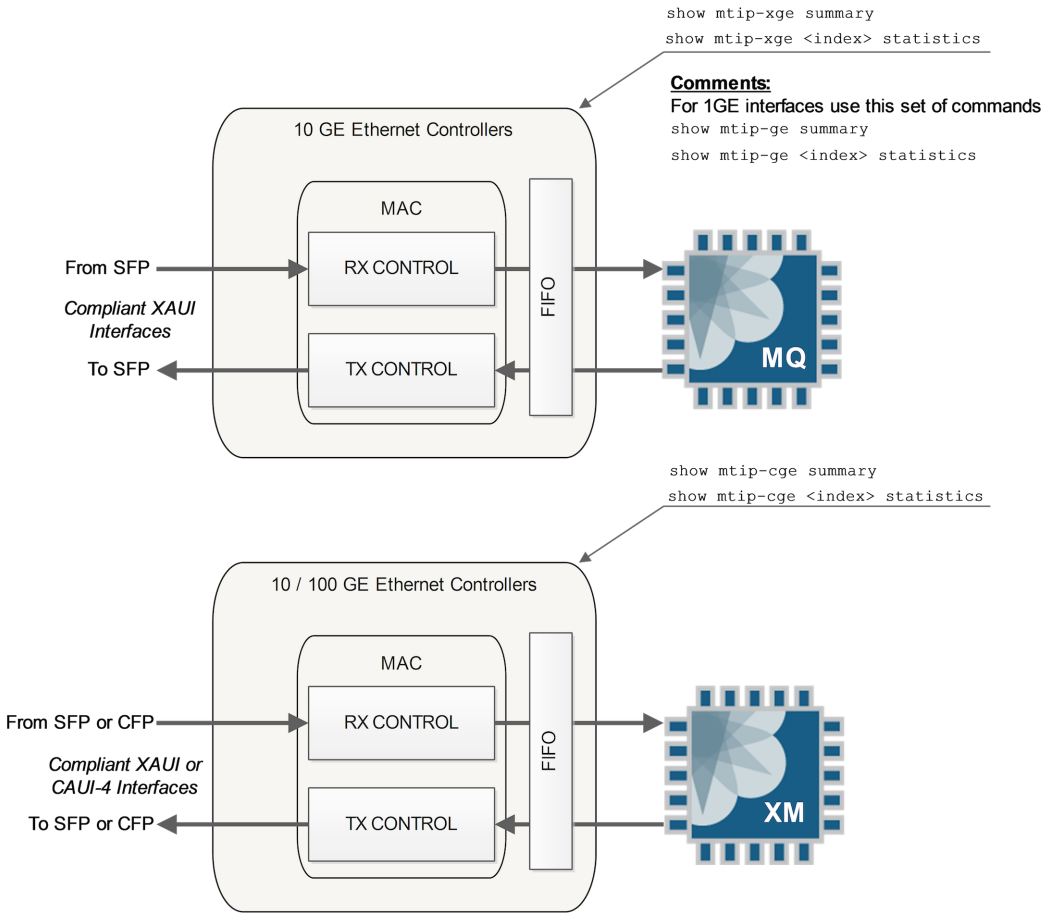



Figure 2.2 10/100 GE Ethernet Controllers

You can retrieve some useful statistics from this controller by using one of the show commands noted in Figure 2.2. First, you need to identify which controller manages the interface by using the `summary` keyword. Then display the statistics with the second command, here shown with a sample output for interface xe-11/0/2.

NOTE Apart from interacting directly with the line card shell, you can also use the `request pfe execute target fpcX` command at the CLI level, allowing you to use `| match` or `except` to filter specific patterns:

```
user@R2-re0> start shell pfe network fpc11

NPC11(R2 vty)# show mtip-xge summary
 ID mtip_xge name        FPC PIC ifd                (ptr)
--- ------------------- --- --- ------------------- --------
  1 mtip_xge.11.0.16      11   0                     548915b8
  2 mtip_xge.11.1.17      11   1                     54891528
  3 mtip_xge.11.2.18      11   2                     54891498
  4 mtip_xge.11.3.19      11   3                     54891408
  5 mtip_xge.11.0.0       11   0 xe-11/0/0           548912e8
  6 mtip_xge.11.0.1       11   0 xe-11/0/1           548911c8
  7 mtip_xge.11.0.2       11   0 xe-11/0/2           54891018 <<<< Our incoming
                                                              interface
[...]

NPC11(R2 vty)# show mtip-xge 7 statistics < Index 7 is the controller of xe-11/0/2
Statistics
----------
  aFramesTransmittedOK:                                    96
  aFramesReceivedOK:                                  2466781
  aFrameCheckSequenceErrors:                            46365
  aAlignmentErrors:                                         0
  aPAUSEMACCtrlFramesTransmitted:                           0
  aPAUSEMACCtrlFramesReceived:                              0
  aFrameTooLongErrors:                                      0
  aInRangeLengthErrors:                                     0
  VLANTransmittedOK:                                        0
  VLANReceivedOK:                                           0
  ifOutOctets:                                           6144
  ifInOctets:                                      1262991872
  ifInUcastPkts:                                      2466781
  ifInMulticastPkts:                                        0
  ifInBroadcastPkts:                                        0
  ifInErrors:                                           46365
  ifOutErrors:                                              0
  ifOutUcastPkts:                                           0
  ifOutMulticastPkts:                                       0
  ifOutBroadcastPkts:                                      96
  etherStatsDropEvents:                                     0
  etherStatsOctets:                                1286730752
  etherStatsPkts:                                     2513146
  etherStatsJabbers:                                        0
  etherStatsFragments:                                      0
  etherStatsUndersizePkts:                                  0
  etherStatsOversizePkts:                                   0
  etherStatsPkts64Octets:                                   0
  etherStatsPkts65to127Octets:                              0
  etherStatsPkts128to255Octets:                             0
  etherStatsPkts256to511Octets:                             0
  etherStatsPkts512to1023Octets:                      2513146  <<<< Some
  etherStatsPkts1024to1518Octets:                           0  stats per Packet Size
  etherStatsPkts1519toMaxOctets:                            0
```

```
etherStatsPkts64OctetsTx:                                96
etherStatsPkts65to127OctetsTx:                            0
etherStatsPkts128to255OctetsTx:                           0
etherStatsPkts256to511OctetsTx:                           0
etherStatsPkts512to1023OctetsTx:                          0
etherStatsPkts1024to1518OctetsTx:                         0
etherStatsPkts1519toMaxOctetsTx:                          0
```

This last command is sometimes useful to track specific packet sizes or specific MAC errors.

## Pre-classifying the Packets (Ingress MQ/XM)

The two flows are then delivered to the MQ (Memory and Queuing) or XM (next-generation of MQ) chips. These two chips have a pre-classifier engine (also present in the IX chip of low speed GE MICs), which is able to determine the packet type in a very simple manner. Two classes are currently used and these two classes actually correspond to two WAN Internal Streams. A third stream is also available but not used. These streams are called *Physical WAN Input Streams* and have a meaning only within the PFE itself.

- CTRL stream: (*control stream*, also known as *medium stream*) conveys protocol traffic (host-destined or in transit) as well as management traffic (for example, ping).

NOTE At this level the linecard can't figure out if the packet is for the host or for transit. This will be determined during the packet lookup, a task performed later by the LU chip.

- BE stream: (Best Effort stream classified low) conveys all other types of traffic, not identified by the pre-classifier engine as control.

NOTE A third Physical WAN Input Stream is called *RT stream* (Real Time stream classified high) and conveys nothing in current Junos implementation for our purposes.

Because they are not control traffic, the two UDP flows are pre-classified to the BE Physical WAN Input Stream of their incoming interface: xe-11/0/2 for Flow 1, and xe-0/0/2 for Flow 2. You can retrieve some useful statistics from the pre-classifier engine by using the set of commands shown in Figure 2.3, which are the same for MQ- or XM-based cards. They give you only input statistics per physical interface (IFD, interface device). For each interface you find the three specific Physical WAN Input Streams.

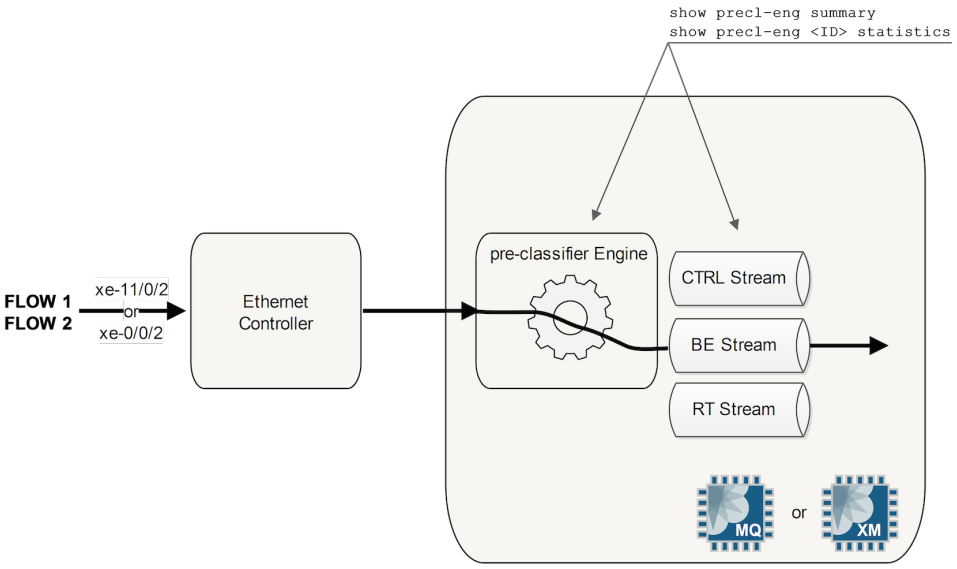



Figure 2.3 The Pre-classifier Engine

Let's have another look again at the xe-11/0/2 side in order to check which Physical WAN Input Streams are associated to this interface and see the statistics of the pre-classifier:

```
NPC11(R2 vty)# show precl-eng summary
 ID  precl_eng name        FPC PIC   (ptr)
--- -------------------- ---- ---  --------
  1 MQ_engine.11.0.16     11   0  547cc708 #precl-eng 1 handles our flow 1 (PIC 0)
  2 MQ_engine.11.1.17     11   1  547cc5a8
  3 MQ_engine.11.2.18     11   2  547cc448
  4 MQ_engine.11.3.19     11   3  547cc2e8
```

NOTE Remember, on 16x10GE MPC cards there are four built-in PICs.

The columns FPC/PIC in the previous command output can help you to identify which pre-classifier engine manages the incoming physical interface. In the case of xe-11/0/2, this is the precl-eng ID 1. (Remember the Juniper interface naming is: xe-fpc_slot/pic_slot/port_num.)

```
NPC11(R2 vty)# show precl-eng 1 statistics
         NPC11(R2 vty)#   show precl-eng 1 statistics
         stream    Traffic
 port      ID       Class        TX pkts            RX pkts          Dropped pkts
------  -------  ----------     ---------          ---------       --------------
  00      1025        RT     0000000000000000  0000000000000000  0000000000000000
  00      1026        CTRL   0000000000000000  0000000000000000  0000000000000000
  00      1027        BE     0000000000000000  0000000000000000  0000000000000000

  01      1029        RT     0000000000000000  0000000000000000  0000000000000000
  01      1030        CTRL   0000000000000000  0000000000000000  0000000000000000
  01      1031        BE     0000000000000000  0000000000000000  0000000000000000

  02      1033        RT     0000000000000000  0000000000000000  0000000000000000
  02      1034        CTRL   0000000000000000  0000000000000000  0000000000000000
  02      1035        BE     0000000002748277  0000000002748277  0000000000000000
                                                 \__FLOW 1
  03      1037        RT     0000000000000000  0000000000000000  0000000000000000
  03      1038        CTRL   0000000000000000  0000000000000000  0000000000000000
  03      1039        BE     0000000000000000  0000000000000000  0000000000000000
```

NOTE Each MQ Chip is connected to four 10GE interfaces. That's why you see 4x3 streams.

Note that the TX and RX columns always show the same values. Inbound streams and pre-classifier are only used for incoming traffic. So, just have a look at the TX column (the output of the pre-classifier, in other words, what the pre-classifier delivers).

Great. As expected, the Flow 1 packets are handled by the BE Physical WAN Input Stream of the xe-11/0/2 physical interface. A similar analysis for interface xe-0/0/2 would confirm the same behavior for Flow 2.

*For PFE troubleshooting you should often know the mapping between the physical port identifier (or IFD – Interface Device) and their associated Internal Physical WAN Input Streams.* Although the previous command that gave you Pre-classifier engine statistics may also give you that information, I prefer to use another PFE command, which helps you to retrieve physical port (IFD) / Physical WAN Input Stream ID mapping. Note the embedded output explanations.

On the MQ chip, the command is:

```
NPC11(R2 vty)# show mqchip 0 ifd <<< On MQ Chip both(Input/output) are displayed

  Input    IFD     IFD       LU
  Stream  Index    Name      Sid  TClass
  ------ ------ ---------- ------ ------
    1033    368  xe-11/0/2     66     hi <<<< RT Stream (unused)
    1034    368  xe-11/0/2     66    med <<<< CTRL Stream
    1035    368  xe-11/0/2     66     lo <<<< BE Stream (Receives our FLOW 1)
    [...]

  Output   IFD     IFD             Base
  Stream  Index    Name     Qsys   Qnum
  ------ ------ ---------- ------ ------
    1026    368  xe-11/0/2    MQ0    512
    [...]
```

On XM Chip, the command is:

```
NPC0(R2 vty)# show xmchip 0 ifd list 0 <<< second 0 means Ingress - 1 Egress

Ingress IFD list
----------------

------------------------------------------------------------------------
IFD name           IFD index    PHY stream    LU SID    Traffic Class
------------------------------------------------------------------------
[...]
xe-0/0/2           340          1033          66        0 (High)
xe-0/0/2           340          1034          66        1 (Medium)
xe-0/0/2           340          1035          66        2 (Low) <<<< BE Stream
                                                        receives the FLOW 2)
[...]
```

Luckily, the Physical WAN Input Stream ID has the same value for the ingress ports of both flows, each in the context of its own MPC. This is a coincidence that simplifies the next step; just remember the number 1035 for both MPCs.

## Creating the Parcel (Ingress MQ/XM)

After the pre-classification processing, the packet is handled by a new functional block of the MQ/XM chip: the WAN Input Block (WI). For each physical interface attached to the PFE, the WI Block receives packets from the three Physical WAN Input Streams (remember only two are used), after the pre-classification. This WI Block stores the packet for future processing in the packet buffer and generates a parcel by catching the first part of each packet.

Let's break for a minute and present the three kinds of data managed internally by the 3D chipsets:

- Parcel: This is actually a chunk (also known as a *first segment*) of the real packet. This chunk contains all the packet headers and some other internal fields. The parcel has a variable length but also a maximum size of 320 bytes. Actually, if the packet size is less than 320 bytes the entire packet is taken into the parcel. Otherwise, if the packet size is above 320 bytes, only the first 256 bytes make it to the parcel.
- Additional segments/chunks corresponding to the data that is not in the parcel are stored in the MQ/XM on-chip and off-chip memories.
- The Cells: When the entire packet needs to move from one PFE to another PFE through the fabric this packet is split in small cells that have a fixed size (64 bytes).

Let's analyze statistics of the WI Block and try to retrieve the 1000 pps of each flow. The "WI" PFE Block doesn't maintain per stream statistics by default. For trouble-shooting purposes, you can enable or disable them for a given stream, here the Physical WAN Input Stream 1035 (BE stream of both interfaces xe-11/0/2 and xe-0/0/2 at their respective MPCs). There are some differences between the MPC 16x10GE and MPC4e cards, which will be discussed throughout this book when appropriate.

Let's activate WI accounting for Physical WAN Input Stream 1035 on the MQ-based card:

```
NPC11(R2 vty)# test mqchip 0 counter wi_rx 0 1035  <<<< first 0 is PFE_ID, second 0 is counter 0
```

Then display WI statistics :

```
NPC11(R2 vty)# show mqchip 0 counters input stream 1035
WI Counters:
            Counter       Packets     Pkt Rate            Bytes    Byte Rate
---------------------------------- ------------ ---------------- --------
RX Stream 1035 (011)         8402         1000          4285020   <<< FLOW 1
[...]
```

Great! You can see the 1000 pps of Flow1. *Don't forget to deactivate* the WI accounting for the Physical WAN Input Stream on the MQ based card:

```
NPC11(R2 vty)# test mqchip 0 counter wi_rx 0 default
```

Do the same for the XM-based card. Activate WI accounting for Physical WAN Input Stream 1035:

```
NPC0(R2 vty)# test xmchip 0 wi stats stream 0 1035 <<< first 0 is PFE_ID, second 0 is counter 0
```

Then display WI statistics. This command gives a lot of information (truncated here), just have a look at the "Tracked Stream Stat" tab:

```
NPC0(R2 vty)# show xmchip 0 phy-stream stats 1035 0 <<< second 0 means Input Direction
WI statistics (WAN Block 0)
---------------------------
[...]
Tracked stream statistics
-------------------------------------------------------------------------------
Track Stream Stream Total Packets        Packets Rate         Total Bytes
      Mask   Match                      (pps)
-------------------------------------------------------------------------------
0     0x7f   0xb    120484              1000 << FLOW 2       61446840
 \___ <<< Track 0 = Counter 0
[...]
```

You can see the 1000 pps of Flow 2. *Don't forget to deactivate* WI accounting for the Physical WAN Input Stream on the XM based card:

```
NPC0(R2 vty)# test xmchip 0 wi stats default 0 0 <<< the second 0 means WAN Group 0 – see Chapter 1
– the third 0 is the counter Index (Track 0 on previous command)
```

Let's move on to the life of the two flows. WI has generated the Parcel so the next step is packet lookup. The Parcel is sent to the LU (Lookup Unit) chip via the "LO" (LU Out) functional block of the MQ or XM Chip. Why just the Parcel? Well, the Parcel contains all the packet headers, and this is enough information to perform route lookup, advanced packet handling, header modification, etc. There is simply no need to send the whole original packet up to the LU chip, so just the Parcel is enough.

The MQ or XM chip adds a header to the Parcel called the *M2L header* (MQ to LU). This header includes some information collected by the MQ or XM chip, like the Physical WAN Input Stream value (in this case 1035). The other packet segments made by WI are pre-buffered in MQ/XM on-chip memory. The MQ/XM off-chip memory will be used later, during the queuing phase.

The LU chip is split internally in several Packet Processor Engines (PPE). The traffic inside the LU chip is load balanced between all the PPEs. An LU chip's PFE-based commands are quite "tricky," so for the sake of this example, let's consider the LU chip as a Black Box that carries out tasks such as packet lookup, traffic load balancing, uRPF check, packet classification, packet filtering, packet accounting, packet policing, and many others that are not covered in this book.

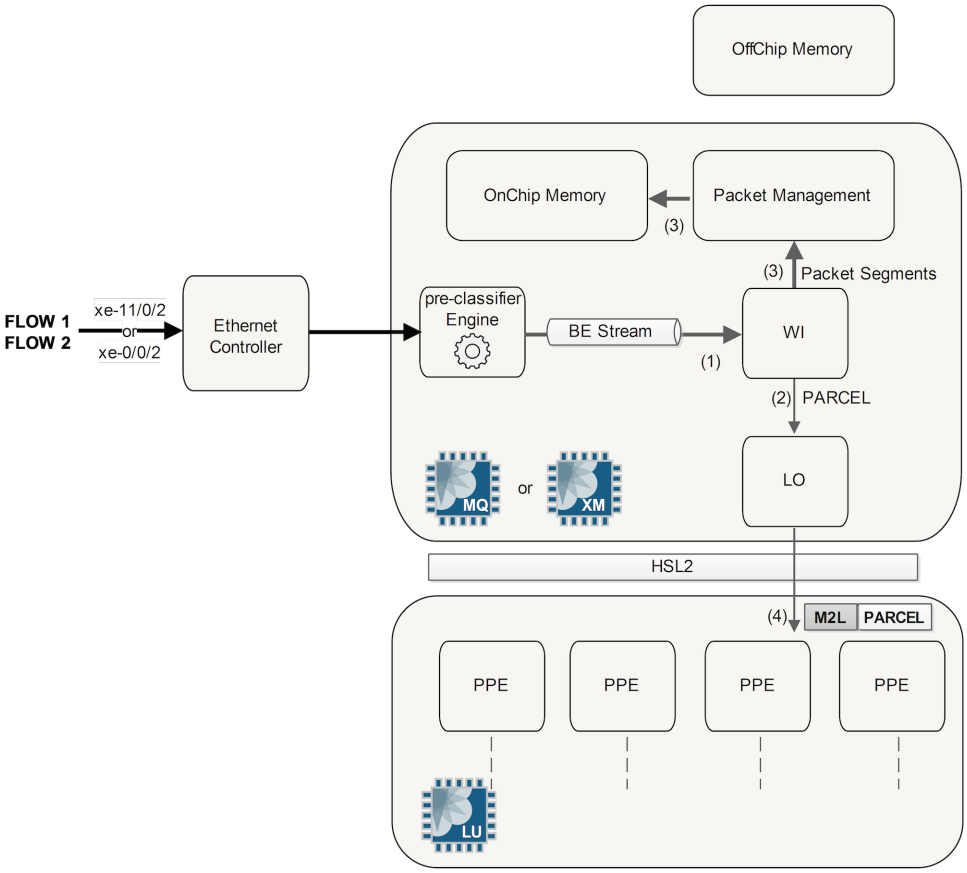



Figure 2.4 From MQ/XM Towards the LU Chip

NOTE MPC4e has two LU chips per XM chip. The XM chip load balances parcels across the two LU chips.

## Forwarding Lookup (Ingress LU)

The LU chip processes the parcel and then performs several tasks depicted in Figure 2.5 carried out by the LU when it receives the parcels of the two flows.

Once it receives the parcel from the MQ or XM "LO" block, the LU chip first extracts the Physical WAN Input Stream ID from the M2L header. Then the parcel is dissected in order to check which protocol conveys the Ethernet frame. In our example the Ethernet type field is equal to 0x800 = IPv4. During the IPv4 sanity check the packet total length and the checksum are compared to the values carried in the IP header. If the sanity check fails, the packet is marked as "to be dropped." (Drops will be performed by the MQ or XM chip – see Chapter 3 for more details).

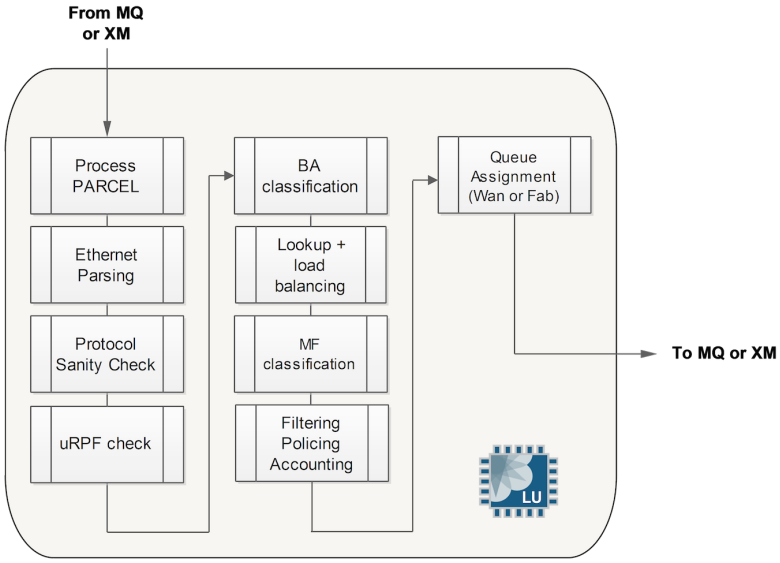



Figure 2.5 LU Next-hop Chaining

The next step is route lookup and the result for each of the example flows is: *packet must be forwarded to another PFE to reach the destination*. Indeed, there are several possible results for a transit packet: the forwarding next hop may be attached to the same PFE as the incoming interface (intra-PFE forwarding), or to a remote PFE (inter-PFE forwarding). For the two flows, this second type of forwarding is involved.

NOTE This book does not specifically cover the case of intra-PFE forwarding. Nevertheless by covering in detail the inter-PFE forwarding you should have all the keys to understand intra-PFE, which is actually a subset of inter-PFE. Forwarding between interfaces attached to the same PFE is typically handled locally and does not involve the fabric.

When the LU chip computes the forwarding next hop, if there is equal-cost multipath (ECMP), or if the outgoing interface is a LAG (Link Aggregated Group), LU also performs the hash during route lookup. In this way, LU is responsible for load balancing traffic across ECMP or LAG's child links. In this book's examples there is no ECMP, but LAG as outgoing interfaces (ae0 for Flow 2 and ae1 for Flow 1, respectively). It was not mentioned previously, but the R2 router is configured with a forwarding table export policy that allows flow-based load balancing at the forwarding level:

```
set policy-options policy-statement load-balancing-policy then load-balance per-packet
set routing-options forwarding-table export load-balancing-policy
```

MORE? The default hash computation is used in this book. You can further fine-tune it by configuring it at the forwarding-options/enhanced-hash-key level.

Okay, let's have a short break in our packet-life analysis and explain a bit more about the "load balancing" function. It is going to be a lengthy diversion, and then we'll return to the MQ/XM chip.

## Load Balancing (Ingress LU)

Flow load balancing is performed by the LU chip. The LU chip extracts from the Parcel some fields of the packet: IP source/destination addresses, for example. Then it computes a hash key based on these fields. The hash is used to select the final and unique forwarding interface, among the several equal-cost choices, in the case of ECMP or LAG.

NOTE This book does not cover the recently added *Adaptive Load Balancing* feature; its functionality allows dynamic load re-balancing of flows depending on their rate.

With the default *static* (non adaptive) load-balancing, some fields are *mandatory* and always used. For example, for IPv4 packets, IP source/destination addresses plus the IP protocol fields are included by default in the hash computation and cannot be removed. Other fields may be added or removed via configuration.

Let's have a look at R2's MPC slot 0:

```
NPC0(R2 vty)# show jnh lb
Unilist Seed Configured 0x919ae752 System Mac address 00:21:59:a2:e8:00
Hash Key Configuration: 0x0000000100e00000 0xffffffffffffffff
           IIF-V4: No  <<< IIF means Incoming Interface index : (the incoming IFL)
         SPORT-V4: Yes
         DPORT-V4: Yes
              TOS: No
      GTP-TEID-V4: No
[...]
```

The fields that are used to compute hash can be retrieved by using the above command. The output doesn't display mandatory fields. Only optional fields which may be added or removed by configuration are displayed.

As you can see, by default for IPv4 traffic, MPC uses only five fields for the hash computation: IP source and destination addresses, IP protocol, and UDP/TCP Source and Destination ports.

Let's move back to our case study: suppose the packet (Parcel) is still in the LU chip and route lookup, plus hash computation has already been performed and the final next hop found. The question is now: *How to check the result of the hash computation performed by the ingress LU chip,* and more importantly, *which output interface among the LAG child links has been selected as the forwarding next hop*?

In other words, which physical interface of AE0 will the Flow 2 be forwarded on, and which physical interface of AE1 will the Flow 1 be forwarded on?

At the PFE level you can find a nifty little command that should answer this question. Actually, it's a tool, called *jsim*. It allows you to craft a Parcel including the fields used by the hash computation, and once the fake Parcel is created you can then run a packet lookup simulation which will give you the forwarding next hop. The jsim tool is a bit tricky because you need to know some information about the packet as well as the "m2l" header in advance. But once you know how to use it, jsim simulates the complete next hop chain resolution including all the tasks performed by the LU chip. What a tool!

CAUTION Again, jsim is shell-based so it is absolutely not supported. Only use it in a production network if instructed to do so by JTAC!

## Step-by-Step on How to Use jsim

Let's try to find the forwarding next hop of the Flow 1.

*Step 1: Open a shell session to the Incoming MPC – the MPC that connects the incoming interface.*

Flow 1 is received by R2 on xe-11/0/2.

```
user@R2> start shell pfe network fpc11
```

*Step 2: Prepare and collect information regarding Jsim configuration.*

Flow 1 is an IPv4 unicast traffic and default load-balancing configuration is in place. So, the fields used for hash computation are:

- IPv4 Source address = 172.16.20.10
- IPv4 Destination address = 172.16.0.3
- IPv4 Protocol = UDP (value is 17)
- UDP Source Port = 1026
- UDP Destination Port = 1026

And you need to collect some internal information:

- The physical Port number on the PIC which receives the stream. Flow 1 is received by the xe-11/0/2 interface, so the physical port number is 2.
- The Physical WAN Input Stream which handles the Flow at MQ or XM Level.

You've previously seen two commands to retrieve this information. One on the MQ Chip:

```
NPC11(R2 vty)# show mqchip 0 ifd
<<< On MQ Chip there is no direction - both(Input/output) are displayed
```

And one on the XM Chip:

```
NPC0(R2 vty)# show xmchip 0 ifd list 0
<<< 2nd 0 means Ingress - 1 Egress
```

In this case, the xe-11/0/2 is hosted by a MQ-based card and Flow 1 is not control or management traffic, so it should be handled by the BE Physical WAN Input Stream. Let's try to retrieve the stream ID:

```
NPC11(R2 vty)# show mqchip 0 ifd
<<< 0 means PFE 0 which hosts the interface

  Input    IFD      IFD        LU
  Stream   Index    Name       Sid  TClass
  ------ ------ ---------- ------ ------
[...]
    1033    490  xe-11/0/2     66     hi
    1034    490  xe-11/0/2     66    med
    1035    490  xe-11/0/2     66     lo <<< lo is the BE stream
[...]
```

So the input physical stream value for the Flow 1 is 1035. Great! You have all that you need to configure jsim now.

*Step 3: Configure the jsim tool.*

Now fill in the collected data into the jsim tool:

```
# Reset Jsim tool
NPC11(R2 vty)# jsim reset

# Tell jsim that the packet is a udp packet (use ? To see the others choices)
NPC11(r2 vty)# set jsim protocol udp

# Fill jsim with packet information used by hash computation
NPC11(R2 vty)# set jsim ipsrc 172.16.20.10
NPC11(R2 vty)# set jsim ipdst 172.16.0.3
NPC11(R2 vty)# set jsim ip-protocol 17
NPC11(R2 vty)# set jsim src-port 1026
NPC11(R2 vty)# set jsim dst-port 1026

# Fill jsim with PFE internal information (Stream ID and Port Number)
NPC11(R2 vty)# set jsim m2l stream 1035
NPC11(R2 vty)# set jsim m2l i2x port_num 2

# Set the my_mac flag. The last bit just tells jsim that the flow was accepted by MAC layer.
NPC11(R2 vty)# set jsim m2l i2x my_mac 1
```

Let's now verify that jsim is configured as expected:

```
NPC11(R2 vty)# show jsim fields
Packet Length :  64

Inport:  WAN
Parcel_type 0 TailEntry 0  Stream Fab 0x40b Off 0
IxPreClass 0 IxPort 2 IxMyMac 1
Ucode protocol:  (Ethernet)
   src_addr: 00:00:00:00:00:00
   dst_addr: 00:00:00:00:00:00
   type: 800
Ucode protocol:  (IPv4)
   src_addr: 172.16.20.10
   dst_addr: 172.16.0.3
   tos: 0
   id: 1
   ttl: 32
   prot: 17
Ucode protocol:  (UDP)
   Source Port : 1026
   Destination Port: 1026
```

*Step 4: Run jsim tool*

It's time to run the jsim tool. It gives you a lot of information (many pages – don't panic). Actually, it simulates the complete next hop chaining which includes, among others, the one depicted in Figure 2.5.

Let's just look at the last step. Note that the actual function name and hexadecimal offset may change across Junos releases. Let's look for function `send_pkt_terminate_if_all_done` in the end:

```
NPC11(R2 vty)# jsim run 0  <<< 0 means PFE 0 (run Jsim on PFE that hosts the incoming interface)
[...]
131  send_pkt_terminate_if_all_done_2 @ 0x0358
Cond SYNC XTXN REORDER_TERMINATE_SEND(PA 0x8641a1c6, 0x00000054)
Packet (h_h 26 @ 0x1c6 h_t 50 @ 0x54):
    04bfe00011f00000
    0001200001750001
    0001000000000000
    0378            <<< This is the forwarding next hop ID in Hexadecimal.
        450000320001
    00001f112f8dac10
    140aac1000030402
    0402001e8b800000
    0000000000000000
    0000000000000000
    00000000
```

*Step 5: Resolve the next hop ID*

To finish up you need to resolve the forwarding next hop ID 0x0378. To do this use this command:

```
NPC11(R2 vty)# show nhdb id 0x0378
   ID     Type     Interface    Next Hop Addr    Protocol      Encap     MTU            Flags
-----  --------  -------------  ---------------  ----------  ------------  ----  ------------------
  888   Unicast  xe-0/0/0.0    172.16.20.6          IPv4     Ethernet  8986  0x0000000000000000
```

Great! You've finally found the child interface of AE1 that should convey the Flow 1 stream. Just verify your work by checking the output statistics of the AE1 child interfaces:

```
user@R2> show interfaces xe-0/0/[0-1] | match "physical|rate"
Physical interface: xe-0/0/0, Enabled, Physical link is Up
  Input rate     : 0 bps (0 pps)
  Output rate    : 3953136 bps (1000 pps)
Physical interface: xe-0/0/1, Enabled, Physical link is Up
  Input rate     : 0 bps (0 pps)
  Output rate    : 0 bps (0 pps)
```

As you can see xe-0/0/0 is the forwarding next hop of Flow 1. If you do the same for Flow 2 you will find interface xe-11/0/1 as the forwarding next hop.

*Step 6: Clean up your test*

Don't forget to clean up your test. Indeed, the jsim's result is saved in the MPC:

```
# First, find the "Idx" of your test. The one called "send_pkt_terminate_if_all_done_2"
NPC11(R2 vty)# show ttrace
 Idx PFE ASIC PPE Ctx Zn   Pending     IDX/Steps/Total  FLAG  CURR_PC  Label
  0   0    0   1  15  23               94/   93/ 1000  SAVE  0x0358   send_pkt_terminate_if_all_
   \___ Idx of your test

# Delete the test result
NPC11(R2 vty)# bringup ttrace 0 delete <<< 0 is the Idx of your test.

# Call back again show ttrace to check that test has been well deleted.
NPC11(R2 vty)# show ttrace
```

## Packet Classification (Ingress LU)

Now that you have found the final forwarding interface by using the jsim tool, remember the packet is still in the LU chip of the Ingress PFE. So the trip within the MX hardware still has a long way to go.

As illustrated in Figure 2.5, the LU chip performs many tasks. It has found the final next hop, and now, the LU chip performs another important task – *packet classification*.

Indeed, the ingress linecard is in charge of classifying the inbound packets and assigning them to the right *forwarding class* and *drop/loss priority*. The LU chip assigns these values but it does not perform any queuing or scheduling. These tasks are performed by the MQ/XM chip instead (details to come).

R2's configuration does not have any explicit CoS BA classifiers applied to the ingress interfaces, so the packets are classified according to the default IP precedence classifier, called **ipprec-compatibility**:

```
user@R2> show class-of-service interface xe-11/0/2.0 | match classifier
Classifier              ipprec-compatibility    ip                    <index>

user@R2> show class-of-service interface xe-0/0/2.0  | match classifier
Classifier              ipprec-compatibility    ip                    <index>

user@R2> show class-of-service classifier name ipprec-compatibility
[...]
000                FC0                                 low
[...]
111                FC3                                 high
[...]
```

- Flow 1 packets have IP Precedence 0, so they are classified to forwarding class FC0 and loss priority (PLP) low.
- Flow 2 packets have IP Precedence 7, so they are classified to FC3 and PLP high.

MORE? The forwarding class and loss priority concepts are explained in detail in *Day One: Junos QoS for IOS Engineers*. See: http://www.juniper.net/us/en/training/jnbooks/day-one/fundamentals-series/junos-qos/.

The forwarding class assignment determines the egress queue where the packet will ultimately be placed on the egress PFE. This is only relevant when the packet is queued towards the egress port facing the outside world, and there is still a long way to go to get there. Let's see how forwarding classes and queue numbers are linked together:

```
user@R2> show configuration class-of-service
forwarding-classes {
    queue 0 FC0 priority low;
    queue 1 FC1 priority low;
    queue 2 FC2 priority low;
    queue 3 FC3 priority high;
}
```

The low/high values on the right have nothing to do with the drop/loss priority (PLP) concept mentioned earlier (although they match this time, it's simply a coincidence). Instead, they refer to the fabric priority. In this scenario of *inter-PFE forwarding*, the LU chip of the ingress linecard not only assigns the packet to the right forwarding

class/drop priority but also to the right fabric *queue* depending on the CoS configuration. Let's wrap up:

- Flow 1 packets are classified to forwarding class FC0 and packet loss priority (PLP) low. If they need to go through the fabric, they will be placed in the low priority fabric queues. If they need to be sent out of a WAN egress port, they will be placed on queue 0.
- Flow 2 packets are classified to forwarding class FC3 and packet loss priority (PLP) high. If they need to go through the fabric, they will be placed in the high priority fabric queues. If they need to be sent out of a WAN egress port, they will be placed on queue 3.

NOTE In this book, the term WAN refers to the interface/ports facing the outside world. The fact that these ports are connected to a local or to a wide area network is irrelevant: they are considered as the WAN.

The LU chip did its job: packet analysis, route-lookup, load balancing, packet classification, and most recently, fabric queue assignment.

It's time for the packet, actually the Parcel, to leave the LU chip. The Parcel comes back to the MQ/XM chip through the LU In Block (LI). The Parcel is "rewritten" before returning it to MQ/XM chip. Indeed, LU adds additional information into a Parcel header. In fact, several headers may be added:

- The first header called the *L2M header* is always present. The LU chip provides the MQ/XM chip with some information through this L2M header. The L2M header has a meaning only inside the PFE. It will be removed by the MQ/XM chip after processing. The L2M header conveys the queue assigned by the LU. This queue may be the fabric queue in the case of inter-PFE forwarding (our case); or the WAN Egress Forwarding Queue in case of intra-PFE Forwarding; or, finally, an egress lookup (we'll see that later).
- The second header called the *Fabric Header* (FAB) has a meaning inside and outside the PFE. It allows inter-PFE forwarding. Actually, this header is added only when the packet should go to another PFE in order to reach its forwarding next hop. The Fabric Header conveys, along with other information, the next hop ID resulting from the packet lookup, the forwarding class, and the drop priority assigned by the ingress LU chip.

In this book's example, the forwarding next hop of both Flow 1 and Flow 2 is outside of the ingress PFE. So L2M and FAB headers are both prepended to the Parcel. For each flow, the ingress LU chip provides (encoded in the L2M header) to the ingress MQ/ XM chip the fabric queue ID upon which the packet must be queued before being forwarded to the destination PFE.

## Inter-PFE Forwarding (from Ingress MQ/XM to Egress MQ/XM)

Once the LU chip does its job, it returns the Parcel back to the MQ/XM chip, more specifically to the LI ("LU in") block of the MQ/XM chip. Now the MQ or XM chip knows that the packet should be forwarded to a remote PFE. It also knows the destination PFE thanks to the fabric queue number carried in the L2M header.

## Fabric Queues (Ingress and Egress MQ/XM)

Let's step back for a moment and explore in depth the fabric queue concept.

Indeed, before moving from one PFE to another the packet is first queued by the ingress MQ/XM chip awaiting authorization from the remote PFE.

Regarding fabric CoS, Juniper supports two levels of priority: *high* and *low*. Actually there are two queues per destination PFE: a high priority and a low priority queue. These two levels of priority are used when fabric congestion occurs. In this situation, scheduling occurs based on 95% for High Priority and 5% for Low Priority queue (for more information on Trio CoS refer to Appendix A).

To introduce fabric CoS let's have a look at the following CLI output command. It shows you the fabric queuing statistics (globally from a MPC to another MPC). Here are the statistics for traffic from MPC 0 to MPC 11 and the reverse path. This command consolidates the fabric queue statistics on a per-MPC basis.

```
user@R2> show class-of-service fabric statistics source 0 destination 11
Destination FPC Index: 11, Source FPC Index: 0
 Total statistics:   High priority           Low priority
    Packets:              67571976              84522585
    Bytes  :           34461578824           43029284328
    Pps    :                1000                     0
    bps    :             4080000                   384
 Tx statistics:      High priority           Low priority
    Packets:              67571976              84522585
    Bytes  :           34461578824           43029284328
    Pps    :                1000                     0 <<< FLOW 2
    bps    :             4080000                     0
 Drop statistics:    High priority           Low priority
    Packets:                    0                     0
    Bytes  :                    0                     0
    Pps    :                    0                     0
    bps    :                    0                     0

user@R2> show class-of-service fabric statistics source 11 destination 0
Destination FPC Index: 0, Source FPC Index: 11
 Total statistics:   High priority           Low priority
    Packets:               283097             161632444
    Bytes  :            144250534           82355300176
    Pps    :                    0                  1001
    bps    :                    0               4080384
 Tx statistics:      High priority           Low priority
    Packets:               283097             161632444
    Bytes  :            144250534           82355300176
    Pps    :                    0                  1001 <<< FLOW 1
    bps    :                    0               4080384
 Drop statistics:    High priority           Low priority
    Packets:                    0                     0
    Bytes  :                    0                     0
    Pps    :                    0                     0
    bps    :                    0                     0
```

Interesting, isn't it ? You can see that the traffic of Flow 2, coming from MPC 0 toward MPC 11, is handled by the high priority queue, whereas Flow 1 is handled by low priority queue when it is forwarded through the fabric from MPC 11 to MPC 0. Why? Well, actually, this behavior has been explicitly configured. Remember Flow 2 is marked with the IP precedence 7, whereas Flow 1 keeps the default value of 0. This triggered a different classification at the LU chip, which resulted in a different fabric priority assignment.

By checking R2's previous configuration you can see that the forwarding class *FC3* is assigned to queue 3 with a fabric priority set to *high*, whereas FC0 mapped to queue 0 has a low priority. These last two points explain why Flow 2 is conveyed within the fabric queue with the high priority but Flow 1 is not.

In our case, the LU chip of ingress MPC 11 selects the forwarding class *FC0* (queue 0) and fabric queue *low* for Flow 1, and MPC 0 selects the forwarding class *FC3* (queue 3) and fabric queue *high* for Flow 2.

Once the final next hop has been found and the packet classified, the fabric queue can easily be deduced taking into account the destination MPC slot number and destination PFE number. Figure 2.6 illustrates the simple rule to find the fabric queue number assigned by the ingress LU chip and conveyed in the L2M header to the MQ/XM chip.

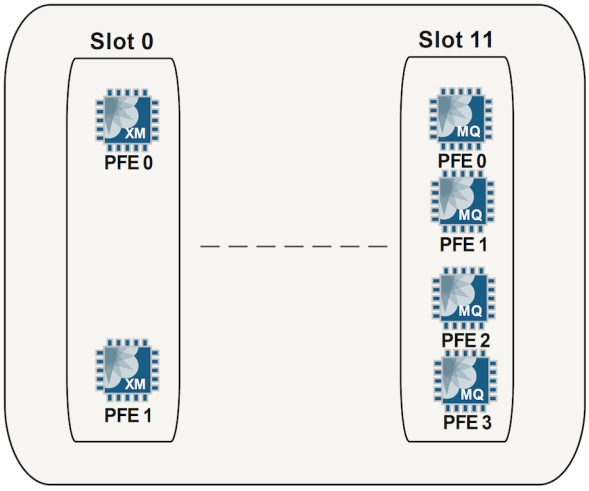



Figure 2.6 How to Compute the Fabric Queue Number Based on the Destination MPC/PFE

Based on this rule you can deduce not only the fabric queue number, also known as the *Output Physical Fabric Stream*, where the ingress PFE chip buffers the packet chunks; but also on which Input Physical Fabric Stream the egress PFE will receive these chunks. We speak about Fabric Queue on ingress MPC because the traffic is really queued there before it's sent through the fabric. On the other hand, the egress MPC receives the packets on a Fabric Stream where there is no real queuing; the packet is received and follows its processing.

Let's do the fabric queue/stream computation as if you were the LU chip. ; )

Let's start with Flow 1. Its packets are received by the MPC in slot 11 PFE ID 0 (xe-11/0/2) and the destination is MPC slot 0 PFE 0 (xe-0/0/0 or xe-0/0/1 (AE1)). Remember Flow 1 was classified in forwarding class FC0, which has a fabric priority set to low by configuration. So Flow 1 will be forwarded to the output fabric queue *low* to reach the (MPC0; PFE0) destination. This output fabric queue number will be:

```
MPC_SLOT_DESTINATION x 4 + PFE_ID = 0 x 4 + 0 = 0
```

It means: *Flow 1 will be queued by the MQ chip 0 of MPC slot 11 at the output fabric queue/stream 0, which has low priority and is pointing to the XM chip 0 of MPC slot 0.*

But Wait! Flow 1, when it later arrives at (MPC0; PFE0), will be received at its input fabric stream number 44. *Why 44 ?*

You have to do the same computation, but from the egress PFE point of view this low priority flow is received from (MPC 11; PFE 0), so the input fabric stream number from the perspective of (MPC0; PFE0) is:

```
MPC_SLOT_SOURCE x 4 + PFE_ID = 11 x 4 + 0 = 44
```

It means: *Flow 1 will be received by the XM chip 0 of MPC slot 0 from the fabric at the input fabric stream 44.*

Let's do the same for Flow 2 as the packets are received by the MPC in slot 0 PFE ID 0 (xe-0/0/2) and the destination is MPC slot 11 PFE 0 (xe-11/0/0 or xe-11/0/1 (AE0)). Flow 2 is classified in forwarding class FC3 with a fabric priority set to *high*. So Flow 2 will be handled at (MPC0; PFE0) by the fabric queue *high* pointing to the (MPC11 ; PFE0) destination. This output fabric queue number is 172:

```
128 + (MPC_SLOT_DESTINATION  x 4 + PFE_ID) = 128 + (11 x 4 + 0) = 172
```

It means: *Flow 2 will be queued by the XM chip 0 of MPC slot 0 at the output fabric queue/stream 172, which has high priority and is pointing to the MQ chip 0 of MPC slot 11.*

But Wait Again! Flow 2, when it later arrives at (MPC11 ; PFE0) will be received at its input fabric stream number 128. Why 128? Again, do the computation but from the egress PFE point of view. This *high* priority flow is received from (MPC 0; PFE 0), so the input fabric stream number from the perspective of (MPC11; PFE0) is 128:

```
128 + (MPC_SLOT_SOURCE x 4 + PFE_ID) = 128 + (0 x 4 + 0) = 128
```

It means: *Flow 2 will be received by the MQ chip 0 of MPC slot 11 from the fabric at the input fabric stream 128.*

Great job if you followed all this. It's a little tricky, so look at Figure 2.7 which is a graphical representation of how flows are conveyed internally in the fabric.

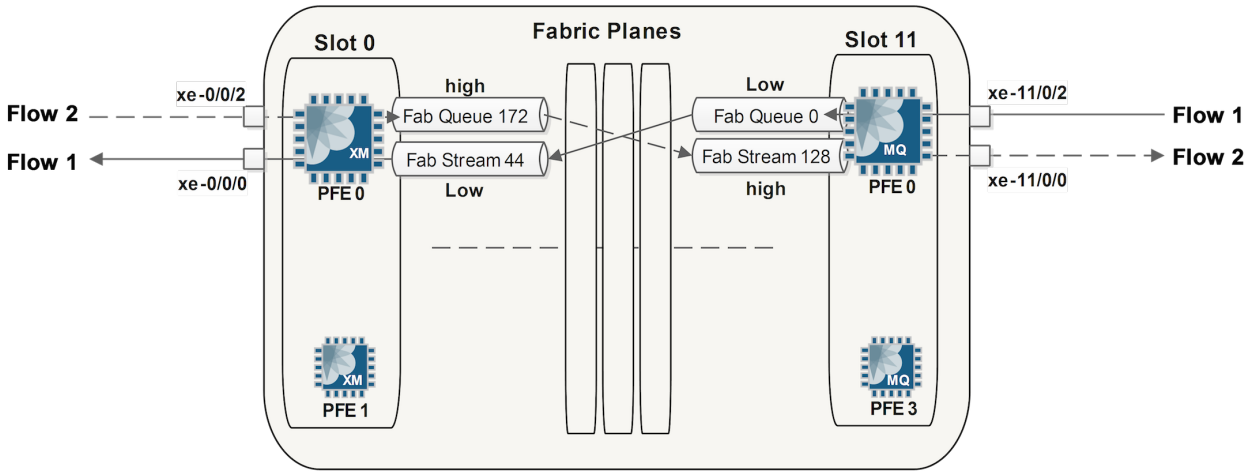



Figure 2.7 Fabric Queues for Flow 1 and 2

## Sending Traffic Through the Fabric

As you have seen, the ingress MQ/XM chip knows the fabric queue number where the packet (Parcel and additional data units) needs to be placed. This is thanks to the L2M (LU to M) header. Well, the MQ/XM chip strips this L2M header after processing it. Only the fabric header is kept and ultimately forwarded to the remote (egress) PFE. This fabric header carries useful information like the final next hop ID resulting from the route lookup, the forwarding class, and the drop priority.

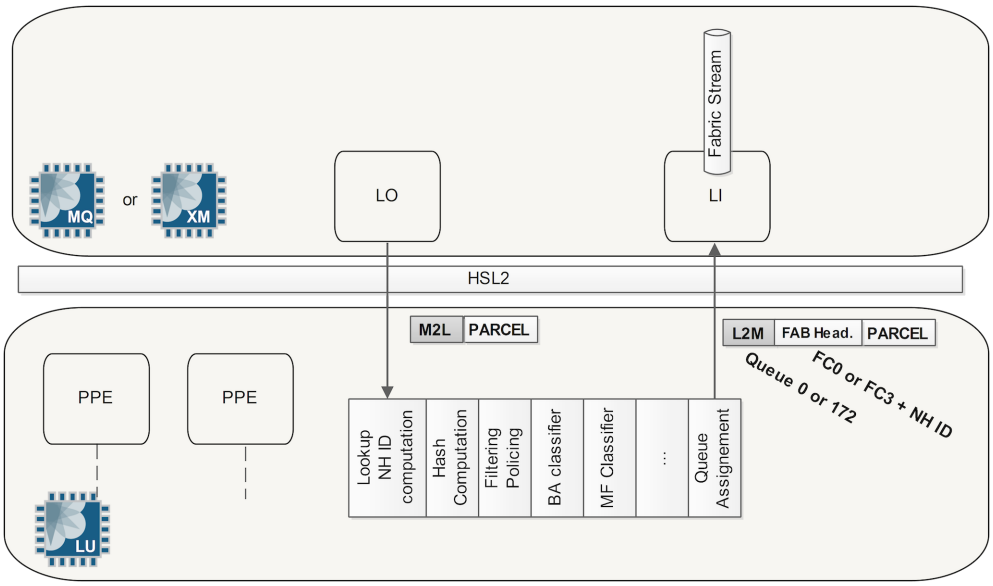



Figure 2.8 From LU back to the "M" Chip

The LI Block at MQ/XM moves the Parcel and the associated segments (previously stored at on-chip memory) to the off-chip memory. Only the *pointer* referring to the real packet is now stored at the MQ or XM on-chip memory. But keep in mind this pointer refers to the packet.

IMPORTANT Only pointers are queued. Real data stays in off-chip memory until it has to be forwarded towards either a remote PFE through the fabric (inter-PFE) or to a WAN interface (intra-PFE).

Now packets are queued and packet scheduling takes place. This is the job of the SCHED block of the MQ or XM chip and it is in charge of providing CoS functions for packets in transit (for control plane packets see Chapter 3).

The SCHED block is complex and a dedicated tangent is required regarding the CoS implementation in MQ or XM chips. Check Appendix A of this book to fully enjoy this tangent.

REMEMBER Each PFE (MQ or XM chip) has two fabric queues (*low* and *high*) towards each possible remote PFE.

The two packet flows are queued into the correct fabric queue or fabric stream by the SCHED block of the MQ/XM chip, waiting for the approval of the remote PFE before sending them through the fabric. The new functional block FO (Fabric Out) part of MQ/XM chip sends the entire packet (actually the Parcel with its FAB header generated by LU chip, plus the other packet segments) to the right destination.

Now the fabric conveys only fixed-size packets called *cells*. Each cell is a 64-byte data unit made up of a header and a payload. When a packet needs to be forwarded to a remote PFE, it is first split in several cells.

The MX fabric is made up of several fabric planes. The cells of a given packet are load-balanced over these several planes in a *round-robin* manner. A sequence number is assigned to each cell, to ensure cell re-ordering on the egress PFE side.

NOTE This book does not provide a deep dive of the MX fabric. You can consider each fabric plane as a switching matrix interconnecting all the PFEs.

The group of cells (of a given packet) does not go directly from a source PFE to a destination PFE. For every single cell, the source PFE needs to first send a request through a specific fabric plane to the remote (destination) PFE. In return, the destination PFE acknowledges the request by sending back a reply over the same fabric plane. When the reply is received by the source PFE, the data cell can be sent over the fabric. The request / grant cells implement a flow control mechanism, but they carry no traffic data themselves.

NOTE Actually, request/grant cells are piggy-backed with other data cells but they are logically independent.

This mechanism also prevents PFE oversubscription. If the destination PFE is congested, it silently discards incoming requests. This indirectly puts pressure on the source PFE, which does not see incoming grants and as a result buffers the packet chunks for some time. The outcome may be RED or TAIL drops.

Figure 2.9 summarizes the packet life of Flow 1 and Flow 2 within the ingress PFE.

And before moving onto the egress PFE side, let's do an analysis of the FO block statistics.

MORE? If you follow our CoS tangent in Appendix A, you will learn how to retrieve fabric queue information details with this command: `show cos halp fabric queue-stats <fabric queue/stream>`.

Like the WI block, the MQ/XM chip doesn't maintain statistics for the fabric queues/streams by default. You have to manually enable statistics for a given fabric queue/stream. Let's do it on the two MPCs and try to track Flow 1 on the FO block of MPC 11 and Flow 2 on the FO block of MPC 0.

Now, let's enable the FO counter (choose counter number is 0) for the Fabric Stream 0 (Flow 1 has been assigned to the low priority queue/stream to reach PFE 0 of the MPC 0):

```
NPC11(R2 vty)# test mqchip 0 counter fo 0 0 <<< second 0 is the counter ID,and the third 0 is the
fabric stream
```

And now show the stats:

```
NPC11(R2 vty)# show mqchip 0 fo stats

FO Counters:
Stream  Mask Match         Packets     Pkt Rate      Bytes      Byte Rate          Cells    Cell Rate
----------------------- ---------------- ---------- ---------------- ------------ -----------
FO0 0x1ff 0x000              3906        1001         1990244       510056           FLOW 1>>>>   8001
FO0 0x000 0x000      51776597305       1023  28037921046891       520404     461979582353       8163
```

```
FO1 0x000 0x000              0          0              0          0          0          0
FO1 0x000 0x000              0          0              0          0          0          0
```

You can see the 1000 pps of Flow 1 and you can also see the rate in data cells per second. Remember that cells have a fixed size of 64 bytes and Flow 1 has a packet size of 512 bytes, so there are eight times more cells than Packets.

Don't forget to stop FO accounting:

```
NPC11(R2 vty)# test mqchip 0 counter fo 0 default
```

Now let's do the same on MPC 0 for Flow 2 (fabric stream 172):

```
NPC0(R2 vty)# test xmchip 0 fo stats stream 0 172 (<<< 0 = counter 0 - 172 the Fabric Stream)
NPC0(R2 vty)# show xmchip 0 fo stats

FO statistics
-------------

Counter set 0

    Stream number mask       : 0x3ff
    Stream number match      : 0xac
    Transmitted packets      : 6752 (1000 pps)  <<< Flow 2
    Transmitted bytes        : 3443520 (4080000 bps)
    Transmitted cells        : 54016 (8000 cps)  <<< Data Cells
```

And you can see the 1000 pps rate of Flow 2 and the 8000 cells per second.

Don't forget to stop FO accounting:

```
NPC0(R2 vty)# test xmchip 0 fo stats default 0
```

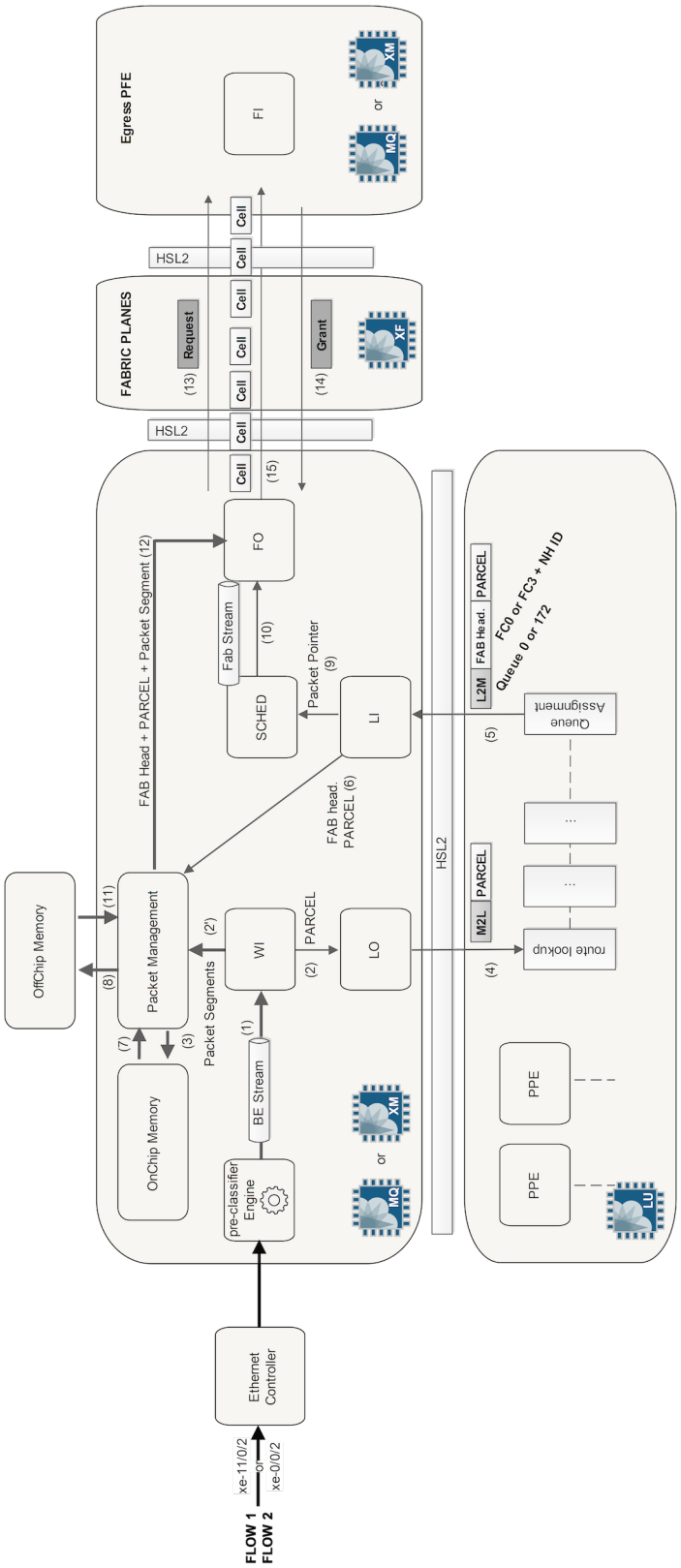



Figure 2.9 Global View of Packet Life – Ingress PFE

## Egress PFE Forwarding

In this section, let's watch the egress PFE which is MPC 0 PFE 0 for Flow 1, and let's also watch MPC 11 PFE 0 for Flow 2. The FI block (Fabric Input) of the egress PFE receives the cells coming from the different fabric planes. FI is in charge of re-ordering the cells of a given packet before the packet is sent to off-chip memory. Only the Parcel, with its FAB header, is sent to the egress LU chip. You can poll the FI block to retrieve statistics as long as these are enabled for a given input Fabric stream.

Flow 1 is now on the FI block of egress MPC 0 PFE 0, but remember Flow 1 comes from the MPC 11 PFE 0 and has a low priority set in the cell header. The low priority Input fabric stream should be: MPC_SOURCE_SLOT x 4 + PFE_SOURCE_ID = 44.

Let's enable FI accounting for Fabric Stream 44 on XM chip 0 of MPC 0, and then show stats:

```
NPC0(R2 vty)# test xmchip 0 fi stats stream 44


NPC0(R2 vty)# show xmchip 0 fi stats

FI statistics
-------------

Stream number mask       : 0x3ff
Stream number match      : 0x2c

--------------------------------------------------------------------------------
Counter Name                    Total                Rate
--------------------------------------------------------------------------------
MIF packet received             4692                 1001 pps  <<< FLOW 1
MIF cells received              37501                7993 cps
MIF packet drops                0                    0 pps
MIF cell drops                  0                    0 cps
Packets sent to PT              4692                 1000 pps
Packets with errors sent to PT  0                    0 pps
--------------------------------------------------------------------------------
```

As you can see, Flow 1 has traveled through the fabric. FI statistics show you the cell rate and packet rate after the cells' re-ordering. Now let's disable FI accounting:

```
NPC0(R2 vty)# test xmchip 0 fi stats default
```

For Flow 2, you must have a look at the FI block of egress MPC 11 (remember Flow 2 comes from the MPC 0 PFE 0 and has a high priority set in the cell header). The high priority Input fabric stream is: 128 + MPC_SOURCE_SLOT x 4 + PFE_SOURCE_ID = 128

Let's enable FI accounting for Fabric Stream 128 on MQ Chip 0 of MPC 11:

```
NPC11(R2 vty)# test mqchip 0 counter fi 128
```

Now let's confirm that Flow 2 is received by egress PFE:

```
NPC11(R2 vty)# show mqchip 0 fi stats

FI Counters:
```

```
  Stream  Mask Match
              0x3ff 0x080          Packets     Pkt Rate            Cells    Cell Rate
------------------------ ---------------- ------------ ---------------- ------------
                Received             7640         1000             61120         8000
                 Dropped                0            0                 0            0
             Pkts to PT              7640         1000
       Errored Pkts to PT                0            0
```

All's well here, so let's disable FI accounting before leaving:

```
NPC11(R2 vty)# test mqchip 0 counter fi default
```

Let's keep following the packets. The Parcel and its FAB header are sent to the LU chip of the egress PFE. As you've seen previously, the MQ/XM chip adds the M2L header before. For "egress" lookup the M2L conveys the Input fabric queue number (also known as *the input fabric stream*): 44 for Flow 1 and 128 for Flow 2.

The egress LU chip performs several tasks. It first extracts information from the M2L and FAB headers. Based on the NH ID (Next Hop ID) and some Parcel fields (like next hop's IP for ARP resolution) it can deduce the forwarding next hop and build the Layer 2 header. This includes pushing or popping headers like MPLS, VLAN, etc., if needed. Moreover, based on the CoS information (forwarding class and loss priority) it can assign the right WAN Physical Output Queue to use and also perform CoS packet re-writing. Once the packet (actually the Parcel) has been re-written, it goes back to the MQ/XM Chip. The FAB Header has been removed and only the L2M header is now present. This header conveys the WAN Output Queue number. Figure 2.10 illustrates this walkthrough.

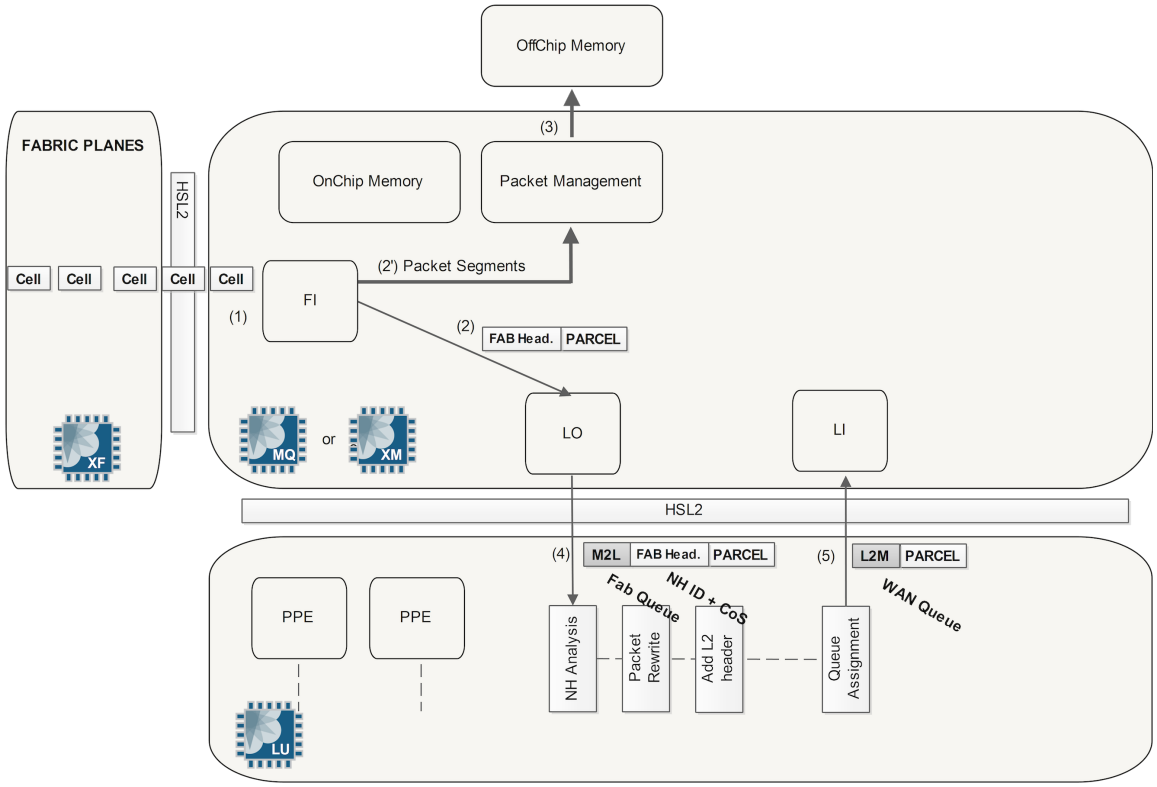



Figure 2.10 Egress PFE – LU Chip Processing

Remember the result of the JSIM tool? Flow 1 should finally be forwarded on the xe-0/0/0 interface and Flow 2 on the xe-11/0/1 interface. Additionally, and based on the packet classification done by the ingress LU chip (FC0 for Flow 1, and FC3 for Flow 2), the egress LU can finally assign the WAN Output Queue number. For Flow 1, it will be the queue 0 of interface xe-0/0/0 and for Flow 2 it will be the queue 3 of interface xe-11/0/1. But these queue numbers are relative, they are not absolute WAN output queue numbers understandable by the MQ/XM chip. For that reason, the LU chip pushes the absolute queue number (uniquely identifying a queue at a specific egress port) in the L2M header, not the relative one.

So, what are these queue values for Flow 1 and Flow 2? You can find the answer on Appendix A, dedicated to WAN CoS: the `show cos halp` command should help you to retrieve these queue values. Let's try it for Flow 1 and Flow 2.

Flow 1 is forwarded on queue 0 of the xe-0/0/0 interface. First of all, find the IFD index assigned to this interface, then execute the above command on MPC0:

```
user@R2> show interfaces xe-0/0/0 | match index
  Interface index: 469, SNMP ifIndex: 526

NPC0(R2 vty)# show cos halp ifd 469
[...]
------------------------------------------------------------------------------------------
Queue  State       Max          Guaranteed    Burst       Weight  G-Pri  E-Pri  WRED  TAIL
Index              Rate         Rate          Size                              Rule  Rule
------------------------------------------------------------------------------------------
256    Configured  222000000    222000000     4194304     62      GL     EL     4     97
257    Configured  555000000    111000000     8388608     25      GL     EL     4     75
258    Configured  555000000    111000000     8388608     25      GL     EL     4     75
259    Configured  555000000    0             8388608     12      GH     EH     4     56
260    Configured  555000000    0             8388608     1       GL     EL     0     1
261    Configured  555000000    0             8388608     1       GL     EL     0     1
262    Configured  555000000    0             8388608     1       GL     EL     0     1
263    Configured  555000000    0             8388608     1       GL     EL     0     1
```

The absolute base queue (relative queue 0) is 256 for this interface. So, egress LU chip of MPC 0 will push the value of 256 as the WAN Output Queue into the L2M header. Let's do the same for the interface xe-11/0/1 which is the egress forwarding interface for Flow 2 (the IFD of xe-11/0/1 is 489):

```
NPC11(R2 vty)# show cos halp ifd 489
[...]
Queue    State         Max       Guaranteed   Burst Weight Priorities Drop-Rules
Index                  rate         rate      size           G    E   Wred  Tail
------ ----------- ------------ ------------ ------ ------ ---------- ----------
   256  Configured    222000000    222000000  32767     62   GL   EL     4   145
   257  Configured    555000000    111000000  32767     25   GL   EL     4   124
   258  Configured    555000000    111000000  32767     25   GL   EL     4   124
   259  Configured    555000000     Disabled  32767     12   GH   EH     4    78
   260  Configured    555000000            0  32767      1   GL   EL     0     7
   261  Configured    555000000            0  32767      1   GL   EL     0     7
   262  Configured    555000000            0  32767      1   GL   EL     0     7
   263  Configured    555000000            0  32767      1   GL   EL     0     7
```

You can see that the absolute base queue (relative queue 0) is also 256 for this interface, but here Flow 2 must be placed in queue 3 (Packets classified in FC3). So, the egress LU chip of MPC 11 will push the value 259 (256 + 3) as the WAN Output Queue in the L2M header.

The parcel moves back to the MQ/XM chip via the LI block (without the FAB

header), then the L2M header is decoded and removed. The MQ/XM knows on which WAN queue of the SCHED block the packet has to be queued. As mentioned previously, the real packet stays in memory and the only information queued is data pointers. Therefore, the MQ/XM pushes the Parcel to off-chip memory, where the remaining packet segments are already.

When the scheduler for the output physical interface decides that the packet can be forwarded (or the pointer is de-queued), it sends a notification to the last functional block WO (WAN Output). This block retrieves packet segments from the off-chip memory, reassembles the packet, and forwards it to the MAC controller to finally leave the router.

NOTE For MQ/XM-based queueing (e.g. queuing management not delegated to QX or XQ chip) the default maximum data buffer value is 100ms. However, the temporal buffer is higher (in ms) with per-unit-scheduler at low shaping rates.

You can now easily confirm that Flow 1 is forwarded via the Queue 0 of the xe-0/0/0 interface, whereas Flow 2 is forwarded via Queue 3 of the xe-11/0/1 interface. The classical `show interface queue` CLI command gives you statistics per queue but this book is a "deep dive" and we can check global WAN statistics directly on the PFE. Let's see how.

Flow 1 is assigned to WAN queue 256 on the XM chip 0 of MPC 0 and Flow 2 to WAN queue 259 on the MQ Chip of MPC 11. The WAN output queue is a q-node (see Appendix A for details). Appendix A also explains that Qsys 0 is for Wan Streams and Qsys 1 is for Fabric Streams. So let's try to retrieve statistics for q-nodes 256 and 259.

For Flow 1 on MPC 0:

```
NPC0(R2 vty)# show xmchip 0 q-node stats 0 256 (2nd 0 mean Qsys 0)

Queue statistics (Queue 0256)
-----------------------------

------------------------------------------------------------------------
Color  Outcome              Counter  Counter  Total                 Rate
                            Index    Name
------------------------------------------------------------------------
All    Forwarded (No rule)  2592     Packets  0                      0 pps
All    Forwarded (No rule)  2592     Bytes    0                      0 bps
All    Forwarded (Rule)     2593     Packets  960405253            999 pps
All    Forwarded (Rule)     2593     Bytes    510934784712    4243032 bps
[...]
```

This output is similar to the classical CLI command:

```
user@R2> show interfaces queue xe-0/0/0 | match "Queue|packets"
Egress queues: 8 supported, 4 in use
Queue: 0, Forwarding classes: FC0
  Queued:
    Packets              :             964231946                  999 pps
  Transmitted:
    Packets              :             960547580                  999 pps
    Tail-dropped packets :                     0                    0 pps
    RL-dropped packets   :                     0                    0 pps
    RED-dropped packets  :                     0                    0 pps
Queue: 1, Forwarding classes: FC1
[...]
```

For Flow 2 on MPC 11:

```
NPC11(R2 vty)# show mqchip 0 dstat stats 0 259 (2nd 0 means Qsys 0)

  QSYS 0 QUEUE 259 colormap 2 stats index 4452:

                 Counter          Packets     Pkt Rate   Bytes          Byte Rate
----------------------- ---------------- ------------ ---------------- -------
    Forwarded (NoRule)            0          0          0              0
    Forwarded (Rule)              542036772  1000       288339755496   532000
```

And this output is similar to the classical CLI command:

```
user@R2> show interfaces queue xe-11/0/1 | match "Queue|packets"
Egress queues: 8 supported, 4 in use
Queue: 0, Forwarding classes: FC0
 [...]
Queue: 1, Forwarding classes: FC1
 [...]
Queue: 2, Forwarding classes: FC2
 [...]
Queue: 3, Forwarding classes: FC3
  Queued:
    Packets              :             542109540                 1000 pps
  Transmitted:
    Packets              :             542109540                 1000 pps
    Tail-dropped packets :                     0                    0 pps
    RL-dropped packets   :                     0                    0 pps
    RED-dropped packets  :                     0                    0 pps
```

Finally, you can check WO statistics, but WO accounting needs to be enabled for a specific WAN Output Stream.

For Flow 1, check the WO Block of XM 0 of MPC 0. First retrieve the WAN Output Stream ID referring to your outgoing interface xe-0/0/0:

```
NPC0(R2 vty)# show xmchip 0 ifd list 1 (<<<< 1 means egress direction)

Egress IFD list
---------------

--------------------------------------------------------------------
IFD name         IFD    PHY     Scheduler  L1 Node  Base   Number of
                 Index  Stream                      Queue  Queues
--------------------------------------------------------------------
xe-0/0/0         469    1024    WAN        32       256    8 <<<< Flow 1 outgoing interface
xe-0/0/1         470    1025    WAN        64       512    8
xe-0/0/2         471    1026    WAN        96       768    8
xe-0/0/3         472    1027    WAN        33       264    8
et-0/1/0         473    1180    WAN        0        0      8
--------------------------------------------------------------------
```

Now let's enable WO accounting for stream 1024:

```
NPC0(R2 vty)# test xmchip 0 wo stats stream 0 1024 (2nd 0 means counter 0)
```

And now let's retrieve WO statistics:

```
NPC0(R2 vty)# show xmchip 0 phy-stream stats 1024 1 (1 means Egress direction)

Aggregated queue statistics
---------------------------
```

```
Queues: 256..263

------------------------------------------------------------------------
Color  Outcome              Counter  Total                Rate
                            Name
------------------------------------------------------------------------
All    Forwarded (No rule)  Packets  0                    0 pps
All    Forwarded (No rule)  Bytes    0                    0 bps
All    Forwarded (Rule)     Packets  1025087991           1000 pps
All    Forwarded (Rule)     Bytes    545304470196         4256000 bps
All    Force drops          Packets  0                    0 pps
All    Force drops          Bytes    0                    0 bps
All    Error drops          Packets  0                    0 pps
All    Error drops          Bytes    0                    0 bps

0      WRED drops           Packets  3684290              0 pps
0      WRED drops           Bytes    1960042280           0 bps
0      TAIL drops           Packets  76                   0 pps
0      TAIL drops           Bytes    40432                0 bps

1      WRED drops           Packets  0                    0 pps
1      WRED drops           Bytes    0                    0 bps
1      TAIL drops           Packets  0                    0 pps
1      TAIL drops           Bytes    0                    0 bps

2      WRED drops           Packets  0                    0 pps
2      WRED drops           Bytes    0                    0 bps
2      TAIL drops           Packets  0                    0 pps
2      TAIL drops           Bytes    0                    0 bps

3      WRED drops           Packets  0                    0 pps
3      WRED drops           Bytes    0                    0 bps
3      TAIL drops           Packets  0                    0 pps
3      TAIL drops           Bytes    0                    0 bps
------------------------------------------------------------------------

CELL destination stream information
-----------------------------------

Byte count     : 0
EOP count      : 0

WO statistics (WAN block 0)
---------------------------

Counter set 0

    Stream number mask       : 0x7f
    Stream number match      : 0x0
    Transmitted packets      : 216815 (1000 pps) <<<< Flow 1
    Transmitted bytes        : 110142020 (4064000 bps)
    Transmitted cells        : 1734520 (8000 cps)
```

You can see there are two kinds of statistics: first, the aggregated statistics of the eight queues attached to the stream (interface), and second, the WO packet statistics. Okay, let's disable WO accounting:

```
NPC0(R2 vty)# test xmchip 0 wo stats default 0 0
<<< the second 0 depends on the virtual WAN System – see Chapter 1 – the third 0 is the counter Index
```

For Flow 2, check the WO Block of MQ 0 of MPC 11. First retrieve the WAN Output

Stream ID referring to your outgoing interface xe-11/0/1:

```
NPC11(R2 vty)# show mqchip 0 ifd
  [...]

  Output   IFD     IFD             Base
  Stream  Index    Name     Qsys   Qnum
  ------ ------ ---------- ------ ------
    1024    488  xe-11/0/0    MQ0      0
    1025    489  xe-11/0/1    MQ0    256 <<<< Flow 2 outgoing interface
    1026    490  xe-11/0/2    MQ0    512
    1027    491  xe-11/0/3    MQ0    776
```

Then enable WO accounting for Output Stream 1025:

```
NPC11(R2 vty)# test mqchip 0 counter wo 0 1025 (2nd 0 means counter 0)
```

And retrieve the WO statistics:

```
NPC11(R2 vty)# show mqchip 0 counters output stream 1025
DSTAT phy_stream 1025 Queue Counters:

  Aggregated Queue Stats QSYS 0 Qs 256..439 Colors 0..3:

  Counter           Packets     Pkt Rate            Bytes    Byte Rate
------------ ---------------- ------------ ---------------- ---------
   Forwarded      17949127027         1000   10398352660362       532000
     Dropped               13            0             6916            0

[...]

WO Counters:
Stream  Mask Match         Packets    Pkt Rate      Bytes    Byte Rate    Cells    Cell Rate
----------------------- ---------- ---------- ---------- ---------- ---------- ----
0x07f 0x001         255246       1000       129664968      508000       2041968       8000
0x070 0x070        8532836          4       575784221         229       16012694          7
```

Okay, let's disable WO accounting:

```
NPC11(R2 vty)# test mqchip 0 counter wo 0 default (<<<< 2nd 0 means counter 0)
```

Figure 2.11 provides you with an excellent graphical view of the packet life in the egress PFE and concludes this chapter regarding unicast traffic.

## Summary

As you've seen, despite the complexity of the MX Series 3D hardware, Junos implements a powerful toolbox to help you to figure out how unicast transit packets are handled and manipulated by the MQ, XM, or LU chips. At any time during packet processing you can have packet and drop statistics, essential tools for advanced troubleshooting. MPLS traffic processing is quite close to the unicast packet flow: only packet lookup is rewritten slightly differently.

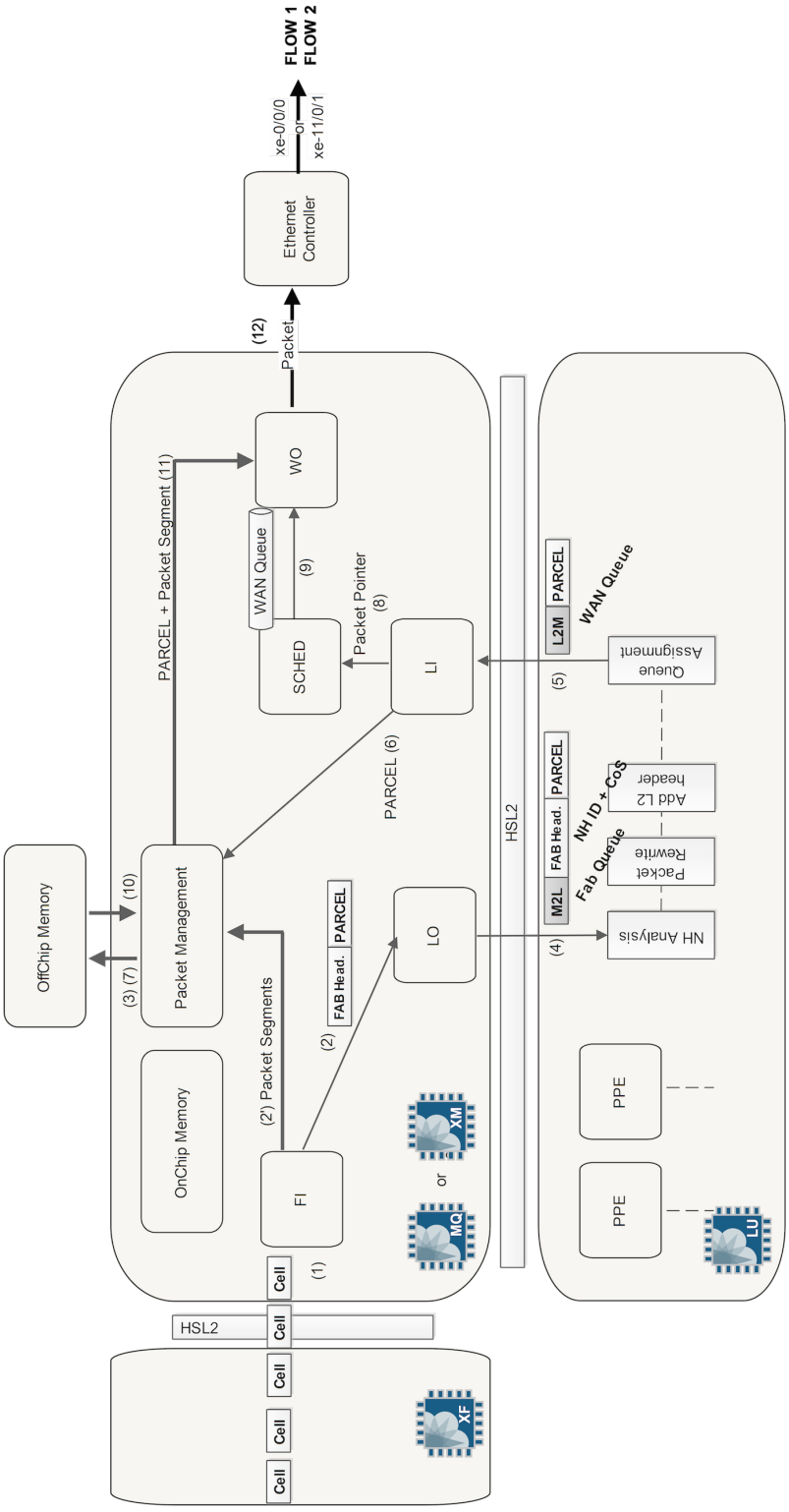



Figure 2.11 Egress PFE – Global Packet Life

# Chapter 3

## On the Way to the Host

This chapter focuses on the life of *exception packets* inside the MX Series 3D linecards. The concept of an exception packet can be understood many different ways, so it's easier to start by explaining the opposite concept first: what is *not* an exception packet?

### *What is not an exception packet?*

- A *transit packet* that is processed only by the forwarding plane, without any intervention of the control plane components (linecard CPU or routing engine CPU). Under normal conditions in real-life production routers, most of the traffic is composed of this type of packets.
- *Internal control packets* between the routing engine CPU and the linecard CPU, that is, exchanged natively through the internal Ethernet switches for internal communication (forwarding table, statistics, line card boot image transfer, etc.). This traffic is not processed by any PFE, only by control plane components.

### *And what is an exception packet?*

- *Input packets* that are discarded by the PFEs and do not reach the control plane.
- *Host inbound packets* that are sent up to the control plane by the ingress PFE.
- *Transit packets with an exceptional condition* (TTL expired, IP Options, a reject action triggering ICMP destination unreachable) or a copy of a true transit packet (firewall log, RE-based sampling).
- *Host-inbound control plane packets*, like a ping to the host, ARP, LACP, routing packet sent to the local host or to a locally-bound 224.0.0.x multicast address.
- *Host outbound packets*: these packets are generated by the control plane and sent to the outside of the MX. They can be either crafted by the routing engine or by the linecard CPU. An example of the latter case is the generation of distributed keepalives (BFD, LACP) and specific ICMP unreachable packets.

### *Who Manages Exceptions?*

Figure 3.1 provides a *macroscopic* view of the possible life of an exception packet. Exceptions can be managed by the LU chip, the linecard's CPU (also known as the µKernel's CPU) or the routing engine's CPU.

Some exception packets are actually discarded or fully processed by the linecard's ASIC, or by the µKernel's CPU, and hence are not punted to the final CPU at the routing engine.

NOTE Remember that WAN is basically "the outside." It can be a LAN or a P2P interface.

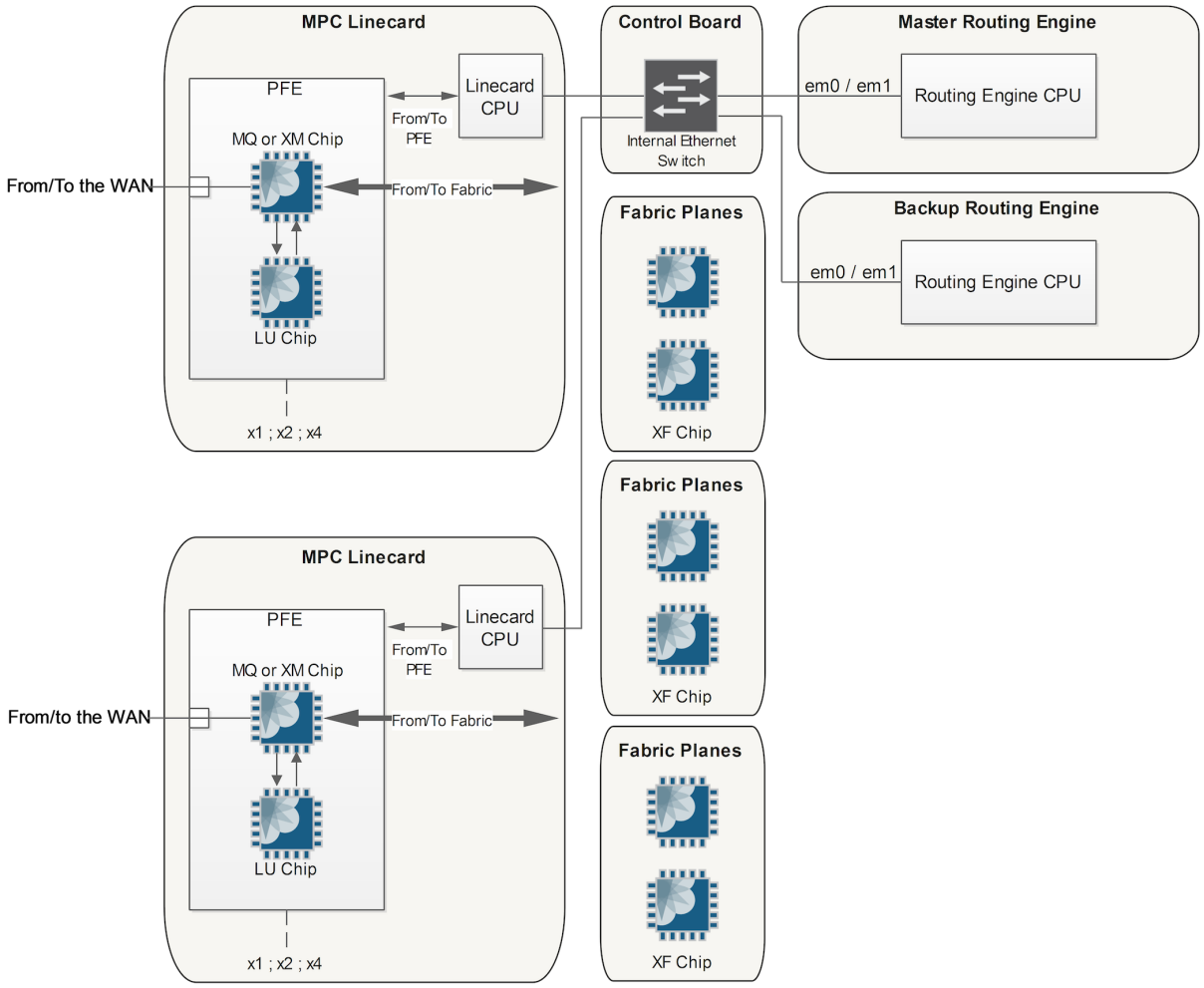



Figure 3.1 Host Inbound Path for Exception Packets

### *The Exceptions List*

The full view of exception packets understood by Junos, can be retrieved by the following CLI command (available in recent releases), which lists all the Input exceptions:

```
user@R2-re0> show pfe statistics exceptions fpc <fpc-slot>

Slot 11

LU 0
Reason                                Type          Packets       Bytes
=====================================================================

Ucode Internal
----------------------
mcast stack overflow                  DISC(33)            0           0
sample stack error                    DISC(35)            0           0
undefined nexthop opcode              DISC(36)            0           0
internal ucode error                  DISC(37)            0           0
invalid fabric hdr version            DISC(41)            0           0
```

```
lu notification                 PUNT(17)          0          0

PFE State Invalid
----------------------
[...]
sw error                        DISC(64)          0          0
invalid fabric token            DISC(75)          0          0
unknown family                  DISC(73)          0          0
unknown vrf                     DISC(77)          0          0
iif down                        DISC(87)          0          0
[...]

Packet Exceptions
----------------------
[...]
bad ipv4 hdr checksum           DISC( 2)          0          0
non-IPv4 layer3 tunnel          DISC( 4)          0          0
GRE unsupported flags           DISC( 5)          0          0
tunnel pkt too short            DISC( 6)          0          0
tunnel hdr too long             DISC( 7)          0          0
bad IPv4 hdr                    DISC(11)          0          0
bad IPv6 hdr                    DISC(57)          0          0
bad IPv4 pkt len                DISC(12)          0          0
bad IPv6 pkt len                DISC(58)          0          0
IP options                      PUNT( 2)          0          0
[...]

Bridging
----------------------
[...]
dmac miss                       DISC(15)          0          0
iif STP Blocked                 DISC( 3)          0          0
mlp pkt                         PUNT(11)          0          0
[...]

Firewall
----------------------
[...]
firewall send to host           PUNT(54)          0          0
firewall send to host for NAT   PUNT(59)          0          0
[...]

Routing
----------------------
[...]
discard route                   DISC(66)          0          0
control pkt punt via nh         PUNT(34)          0          0
host route                      PUNT(32)          0          0
[...]

Subscriber
----------------------
[...]
pppoe padi pkt                  PUNT(45)          0          0
pppoe padr pkt                  PUNT(46)          0          0
pppoe padt pkt                  PUNT(47)          0          0
[...]

Misc
----------------------
[...]
sample syslog                   PUNT(41)          0          0
sample host                     PUNT(42)          0          0
[...]
```

NOTE In earlier Junos releases, you can collect the same information using the `show jnh <pfe-instance> exceptions [terse]` PFE shell command. Some counters can be cleared with `clear jnh <pfe-instance> exceptions`.

This command gives inbound exception statistics per LU instance (or, in other words, per PFE). The output omits some exceptions since the exception list is quite long. It can be shortened with the modifier: `| except "0"`.

CAUTION Remember that a MPC4e card has two LU chips per PFE (per XM Chip). The set of LU chips belonging to the same PFE is called an *LU instance* or *LU complex*. LU instance 0 is PFE 0, and its stats are the sum of LU chip 0 and LU chip 4 stats. LU instance 1 is PFE 1, and its stats are the sum of LU chip 1 and LU chip 5 stats.

The type column in the output provides two interesting pieces of information:

- The Next-Hop type: Discard or Punt next hop. Discard means silently discard the packet at PFE level. Punt means that the packet should be managed by the control plane for further processing.
- The Exception code between ( ): these exception codes are used internally to identify the type of exception. Note that only the combination of next hop type plus the exception code has a meaning. Indeed, the same numerical exception code can be used for two different next hop types. Here's an example:

```
user@R2-re0> show pfe statistics exceptions fpc 11 | match "( 2)"
bad ipv4 hdr checksum              DISC( 2)          0          0
IP options                         PUNT( 2)          0          0
```

Packets are flagged as exceptions by the LU chip. Once the LU microcode has identified a packet as an exception, other post identification mechanisms are used to classify the packet with the right exception next hop. After that, the LU chip can perform other actions like queue assignment, packet filtering, accounting, or policing. Finally, the MQ or XM chip will perform queuing functions towards the host.

NOTE Host here means "the path to the host." In other words, this includes the µKernel's CPU and/or the CPU of the routing engine.

In this chapter, you will follow along analyzing three types of exception traffic:

- A *Layer 3 packet*: A simple ping from R1 ae0 interface's address to the R2 ae0 interface's address that covers the host-inbound and host-outbound paths.
- A *Layer 2 packet*: How MPC handles a simple ARP request.
- A *specific Layer 2 control plane packet*: How MPC handles LACP packets.

Let's start with the Layer 3 packet since it covers more shared concepts.

## The Host Inbound Path: Five Stages

The baseline example used in this book is quite simple: incoming ICMP echo requests, targeted to R2's local IPv4 addresses. Each of these packets follow a fascinating trip in five stages: from MQ/XM to LU; inside the LU; from LU back to MQ/XM; from MQ/XM up to the host; and inside the host (control plane) components.

## Ping to the Host, Stage One: from MQ/XM to LU

Let's embellish our simple network diagram that started with Figure 2.1, and add some IPv4 information regarding the host traffic. In Figure 3.2, you can see that there are actually two active pings. The first one arrives at the MPC 16x10GE side and the second one arrives at the MPC4e side. This is to provide the troubleshooting commands for both kinds of card. Even though both cards share a lot of common commands, sometimes the syntax differs from one to another.

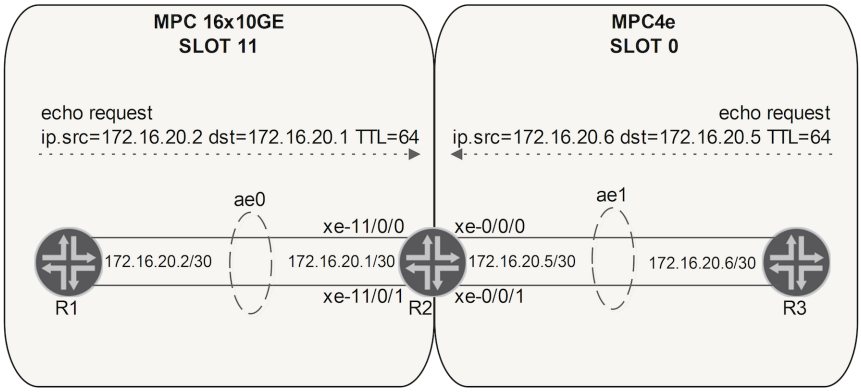



Figure 3.2 Ping to the Host Interfaces

From R1 the command is:

```
user@R1-re0> ping 172.16.20.1 source 172.16.20.2 ttl 64 interval 0.1 pattern "4441594f4e4520"
```

From R3 the command is:

```
user@R3-re0> ping 172.16.20.5 source 172.16.20.6 ttl 64 interval 0.1 pattern "4441594f4e4520"
```

TIP Include a specific pattern to track your packet when it is shown in hexa/ascii mode. The pattern chosen here is actually `DAYONE`.

Let's start to analyze the two pings coming from R1 or R3. The link between routers R1 and R2, or the link between R2 and R3, are aggregated links (respectively ae0 and ae1). Depending on the load balancing algorithm on R1 and R3, the ping from R1 might arrive on the xe-11/0/0 or the xe-11/0/1 interface of R2; and the ping from R3, on the xe-0/0/0 or xe-0/0/1 interface. Let's assume in this example that the ping from R1 enters R2 through the xe-11/0/0 interface and the one from R3 enters R2 through the xe-0/0/0 interface.

As you've already seen in Chapter 2, when a packet enters the router, it is first handled by the MAC controller at the PFE level. Our "echo request" packets are then pre-classified by the MQ or XM chip. As shown next in Figure 3.3, the echo requests are pre-classified in the CTRL stream associated to the incoming interfaces: xe-11/0/0 for pings coming from R1, and xe-0/0/0 for pings coming from R3.

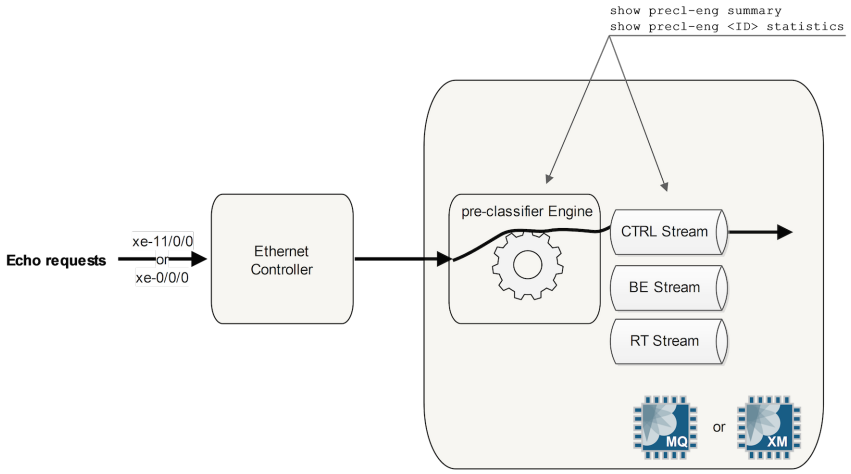



Figure 3.3 Pre-classification of the Incoming ICMP Echo Requests

Let's have a look at the xe-11/0/0 side in order to check which are the Physical WAN Input Streams associated with this interface and see the statistics of the pre-classifier.

```
NPC11(R2 vty)# show precl-eng summary
 ID  precl_eng name      FPC PIC   (ptr)
--- -------------------- ---- ---  --------
  1 MQ_engine.11.0.16    11   0  547cc708 <<< precl-eng 1 handles the echo requests (PIC 0)
  2 MQ_engine.11.1.17    11   1  547cc5a8
  3 MQ_engine.11.2.18    11   2  547cc448
  4 MQ_engine.11.3.19    11   3  547cc2e8
```

NOTE Remember that on 16x10GE MPC cards there are four built-in PICs.

The columns FPC/PIC of the following command help you to identify which pre-classifier engine manages each incoming physical interface. In the case of xe-11/0/0, this is the precl-eng ID 1. Remember, the Juniper interface naming is: *xe-fpc_slot/pic_slot/port_num*.

```
NPC11(R2 vty)# show precl-eng 1 statistics
       stream   Traffic
 port    ID      Class        TX pkts             RX pkts          Dropped pkts
------ ------  --------     ---------          ---------       ------------
  00    1025       RT     0000000000000000   0000000000000000   0000000000000000
  00    1026       CTRL   0000000000000220   0000000000000220   <<<< pings
  00    1027       BE     0000000000000000   0000000000000000   0000000000000000

  01    1029       RT     0000000000000000   0000000000000000   0000000000000000
  01    1030       CTRL   0000000000000000   0000000000000000   0000000000000000
  01    1031       BE     0000000000000000   0000000000000000   0000000000000000

  02    1033       RT     0000000000000000   0000000000000000   0000000000000000
  02    1034       CTRL   0000000000000000   0000000000000000   0000000000000000
  02    1035       BE     0000000000000000   0000000000000000   0000000000000000

  03    1037       RT     0000000000000000   0000000000000000   0000000000000000
  03    1038       CTRL   0000000000000000   0000000000000000   0000000000000000
  03    1039       BE     0000000000000000   0000000000000000   0000000000000000
```

Great! As expected, you can see that the echo requests are forwarded in the CTRL (also known as *Medium*) WAN Input Stream of the xe-11/0/0 physical interface.

Now, let's enter the next functional Block: the WI block. In Chapter 2 the WI (also known as WAN Input) Block role was explained. It receives traffic from the three Input streams of each physical interfaces attached to the PFE. The WI block then stores the packet for future processing in the packet buffer and generates a Parcel by "catching" the first part of each packet.

Let's enforce your understanding of the WI block by collecting WI statistics for the Physical WAN Input Stream 1026 of interface xe-11/0/0. Why 1026 ? Because the above command output states that it is the WAN Input stream ID for CTRL traffic arriving at port xe-11/0/0. Actually, Chapter 2 also provided additional commands to retrieve this information.

On the MQ chip (use "med" stream = CTRL stream) :

```
NPC11(R2 vty)# show mqchip 0 ifd <<< on MQ Chip there is no direction – both
                                      (Input/output) are displayed
```

And on the XM chip (use "med" stream = CTRL S=stream)

```
NPC0(R2 vty)# show xmchip 0 ifd list 0 <<< second 0 means Ingress – 1 Egress
```

Now, enable accounting for this WI Stream on MPC 11:

```
NPC11(R2 vty)# test mqchip 0 counter wi_rx 0 1026 <<< second 0 is the counter 0 (Wi supports 4
                                                    counters max)
```

And then display the corresponding WI statistics at MPC 11:

```
NPC11(R2 vty)# show mqchip 0 counters input stream 1026
WI Counters:
                  Counter      Packets     Pkt Rate            Bytes    Byte Rate
------------------------ ------------ ------------ ---------------- ------------
    RX Stream 1026 (002)            0           10   <<< our pings
    RX Stream 1026 (002)       596381            0         59515056            0
    RX Stream 1027 (003)            0            0                0            0
    RX Stream 1151 (127)      4252703            1        200068178           34

    DROP Port 0 TClass 0            0            0                0            0
    DROP Port 0 TClass 1            0            0                0            0
    DROP Port 0 TClass 2            0            0                0            0
    DROP Port 0 TClass 3            0            0                0            0
```

Now disable WI accounting for this WAN Stream:

```
NPC11(R2 vty)# test mqchip 0 counter wi_rx 0 default
```

Let's do the same for the MPC in slot 0 (XM based). First activate WI accounting for the stream (whose number 1026 is identified by one of the aforementioned XM commands – see details on Chapter 2):

```
NPC0(R2 vty)# test xmchip 0 wi stats stream 0 1026
```

Then display WI statistics for MPC 0 (this command gives a lot of information and is truncated here), just have a look to the "Tracked Stream Stat" tab):

```
NPC0(R2 vty)# show xmchip 0 phy-stream stats 1026 0 <<< second 0 means Input Direction
WI statistics (WAN Block 0)
---------------------------
[…]
```

```
Tracked stream statistics
-------------------------
Track Stream Stream Total Packets  Packets Rate     Total Bytes    Bytes Rate  Total EOPE    EOPE
Rate
     Mask   Match              (pps)           (bps)        (pps)
------------------------------
0    0x7f   0x2    1525          10  <<< Our Pings 152500        8000       0
 \__ Track 0 = Counter Index 0
1    0x7c   0x4    2147483648    0               137438953472  0          0
2    0x7c   0x8    2354225199    0               161385089118  0          0
[...]
```

Okay. Let's disable WI accounting for this WAN Stream on MPC 0:

```
NPC0(R2 vty)# test xmchip 0 wi stats default 0 0 <<< the second 0 depends on the virtual WAN System
                                                     (see Chapter 1), while the third 0 is the counter Index
```

Let's continue to move on with the life of the incoming echo request packet. The next step is packet lookup. The Parcel is sent to the LU chip via the LO functional block of the MQ or XM chip, as shown in Figure 3.4. The "M" (MQ/XM) chip adds a header to the Parcel called the *M2L header*. This header includes information like the Wan Input Stream value: in this case, 1026.

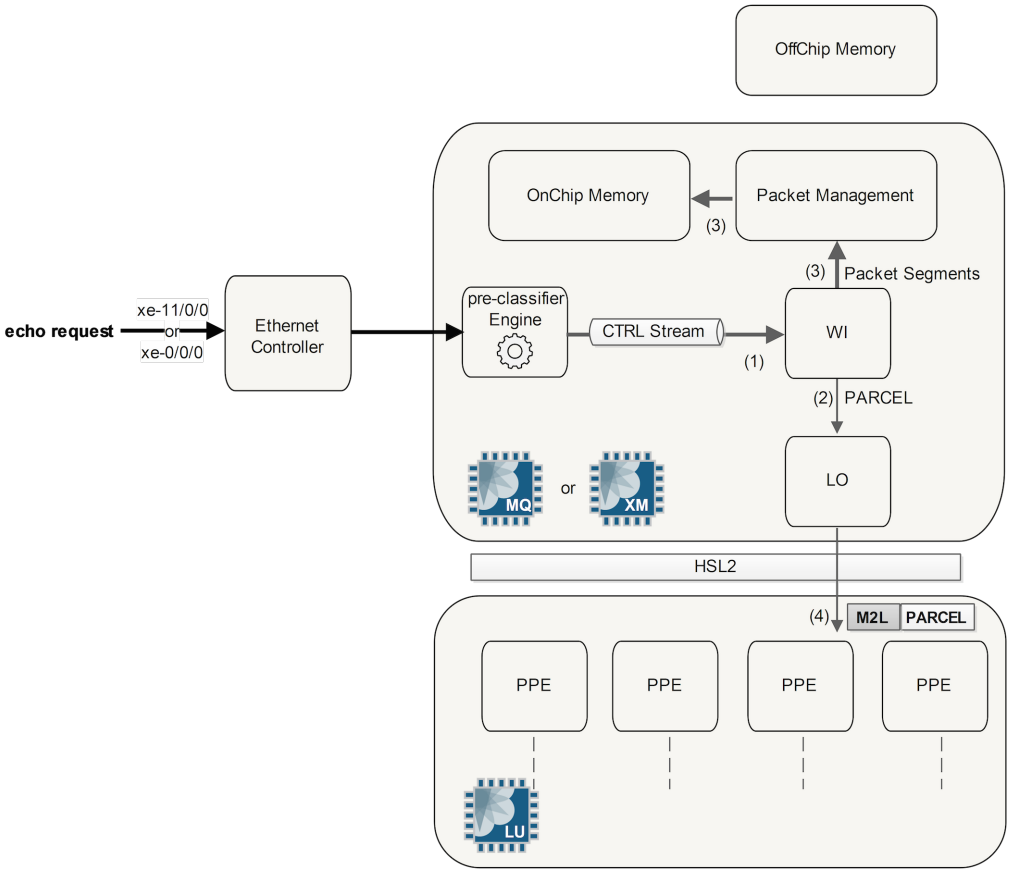



Figure 3.4 From MQ/XM Towards the LU Chip

## Ping to the Host, Stage Two: Inside the LU

As you've seen in Chapter 2, the LU chip processes the Parcel and then performs several tasks. Once the LU chip has identified the packet as host traffic, some specific tasks regarding host inbound traffic are triggered.

Figure 3.5 depicts some of the tasks carried out by the LU chip when it receives the echo request Parcel packet.

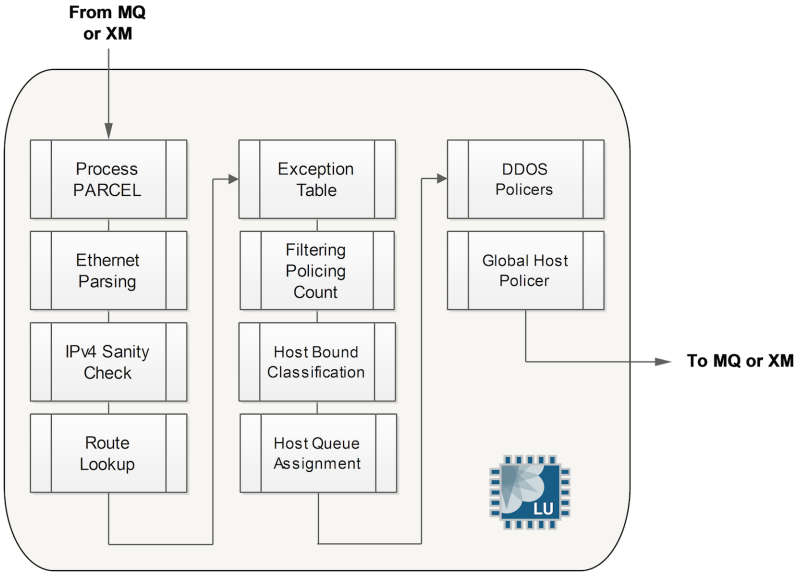



Figure 3.5 LU Lookup Chain for Host Traffic

Once it receives the Parcel from the MQ or XM "LO" block, the LU chip first extracts the Physical WAN Input Stream ID from the M2L header. Then the Parcel is dissected in order to check which protocol conveys the Ethernet frame. In our example the Ethernet type field is equal to 0x800 = IPv4. During the IPv4 sanity check the packet total length and the checksum are compared to the values carried in the IP header. If the sanity check fails, the packet is marked as "To be Dropped" (Drops will be performed by MQ or XM chip –more details later in this chapter).

In this case, the route lookup results in the packet targeting the host. The LU chip tries to match the exception type with its exception table. The ICMP request to the host is a known exception called host route exception (PUNT 32). Remember this value (32). You can see the exception counter named host route incrementing via the following PFE command. Here the jnh instance 0 is checked because both links of R2's ae0 are attached to the PFE 0 of MPC 11 .

```
NPC11(R2 vty)# show jnh 0 exceptions terse
Reason                          Type          Packets       Bytes
================================================================

Routing
----------------------
host route                      PUNT(32)           62        5208
```

For each exception, there is a list of triggered actions, and in this case, the host path protection is performed. The host path is protected first by the loopback firewall filter. In this case, there is a very simple "inet" firewall filter configured on R2 that only rate-limits the ICMP packets to 100Mbits/s and counts them:

```
interfaces {
   lo0 {
      unit 0 {
         family inet {
            filter {
               input protect-re;
            }
            address 172.16.21.2/32 {
               primary;
            }
         }
      }
   }
}
firewall {
   family inet {
      filter protect-re {
         term ICMP {
            from {
               protocol icmp;
            }
            then {
               policer ICMP-100M;
               count ICMP-CPT;
               accept;
            }
         }
         term OTHER {
            then accept;
         }
      }
   }
   policer ICMP-1M {
      if-exceeding {
         bandwidth-limit 100m;
         burst-size-limit 150k;
      }
      then discard;
   }
}
```

The ICMP echo-request is counted and policed by the LU chip. At the PFE level you can easily list the firewall filters available and programed at the ASIC. As you can see, there are default firewall filters named __XXX__ that are always applied depending on the packet's family. The two other default filters named HOST-BOUND will be discussed later in this chapter:

```
NPC11(R2 vty)# show filter
Program Filters:
---------------
   Index     Dir     Cnt    Text     Bss  Name
--------  ------  ------  ------  ------  --------

Term Filters:
------------
   Index    Semantic   Name
--------  ---------- ------
       6  Classic    __default_bpdu_filter__
       8  Classic    protect-re
   17000  Classic    __default_arp_policer__
   57008  Classic    __cfm_filter_shared_lc__
   65280  Classic    __auto_policer_template__
   65281  Classic    __auto_policer_template_1__
```

```
   65282  Classic    __auto_policer_template_2__
   65283  Classic    __auto_policer_template_3__
   65284  Classic    __auto_policer_template_4__
   65285  Classic    __auto_policer_template_5__
   65286  Classic    __auto_policer_template_6__
   65287  Classic    __auto_policer_template_7__
   65288  Classic    __auto_policer_template_8__
16777216  Classic    fnp-filter-level-all
46137345  Classic    HOSTBOUND_IPv4_FILTER
46137346  Classic    HOSTBOUND_IPv6_FILTER
46137353  Classic    filter-control-subtypes
```

Each Firewall has a unique index and you can see the protect-re filter has an index of 8. You can use a second PFE command to see how the firewall filter is programed at the PFE Level:

```
NPC11(R2 vty)# show filter index 8 program
Filter index = 8
Optimization flag: 0xf7
Filter notify host id = 0
Filter properties: None
Filter state = CONSISTENT
term ICMP
term priority 0
    protocol
         1
        false branch to match action in rule OTHER
    then
        accept
        policer template ICMP-100M
        policer ICMP-100M-ICMP
                app_type 0
                bandwidth-limit 100000000 bits/sec
                burst-size-limit 150000 bytes
                discard
        count ICMP-CPT
term OTHER
term priority 0
    then
        accept
```

You can see that the policer is correctly associated to the firewall filter as expected. Now let's see how the policer ICMP-100M-ICMP works. The following CLI command gives you passed and dropped packets:

```
user@R2-re0> show Firewall filter protect-re
Filter: protect-re
Counters:
Name                                                Bytes              Packets
ICMP-CPT                                         31239601               393876
Policers:
Name                                                Bytes              Packets
ICMP-100M-ICMP                                          0                    0
```

A similar PFE command gives you the same results but from the PFE's point of view – remember the CLI command aggregates stats from all PFEs. This PFE command could sometimes be useful to identify which PFE receives or drops packets. The PFE command takes as input the index of the firewall filter, in this case index 8:

```
NPC11(R2 vty)# show filter index 8 counters
Filter Counters/Policers:
   Index               Packets                 Bytes  Name
--------  --------------------  --------------------  --------
       8                118134               9923256 ICMP-CPT
       8                     0                     0  ICMP-100M-ICMP(out of spec)
       8                     0                     0  ICMP-100M-ICMP(offered)
       8                     0                     0  ICMP-100M-ICMP(transmitted)
```

So, our ping traffic is a 10pps stream that easily passes the lo0.0 firewall filter without any issue. Remember that default firewall filters are subsequently applied after the lo0.0 firewall filter. These are applied only if the packet matches the family of the firewall filter.

*So what is the next step?*

Actually, it depends on the *type* of exception. Packet identification requests several tasks. Depending on the type of exception, either more identification tasks or fewer identification tasks are needed.

As previously mentioned, the LU chip maintains an exceptions table and this table is called to identify the exception. In the case of an ICMP echo request, the packet has been flagged as *host route exception* . The LU chip then tries to find more precisely which kind of packet is destined to the host. This brings us to a short tangent in the path of the echo request packet.

## Host Inbound Packet Identification and DDOS Protection

On MPC line cards, a host path protection called *DDOS* (Distributed Denial of Service) *protection* is enabled by default. Each protocol, identified uniquely by a Protocol_ID, is assigned to a dedicated policer.

NOTE Protocol_ID is not based on the protocol field of IP datagrams.

Each protocol policer has a default value which is configurable. And each protocol is rate limited by a set of hierarchical policers. This means that the same policer, assigned to a given Protocol_ID, may be applied at several levels of the host path. Actually there are three levels: the LU chip, the µKernel's CPU, and the routing engine's CPU. Therefore, to reach a *CPU's process/task,* a given exception packet should pass through several policers.

Once the packet has matched an entry in the exception table, deep packet identification is triggered. Depending on the exception, the packet type identification is first managed by a default firewall filter named *HOSTBOUND_IPv4_FILTER* (also known as HBC for HostBound Classification filter). But keep in mind that the HBC filter is not used for every exception (for example, for non-IP Layer 2 PDUs).

The HBC filter only assigns a *ddos-proto ID*. This DDOS_ID is later used internally by the LU chip during the packet identification phase. The final Protocol ID which results from the complete packet identification process will typically be derived directly from the DDOS ID.

In some cases, a more granular analysis is performed. The LU chip will try to compare the packet with a list of known packet types associated to this protocol (DDOS_ID). This depends, once again, on the specifics of the DDOS_ID.

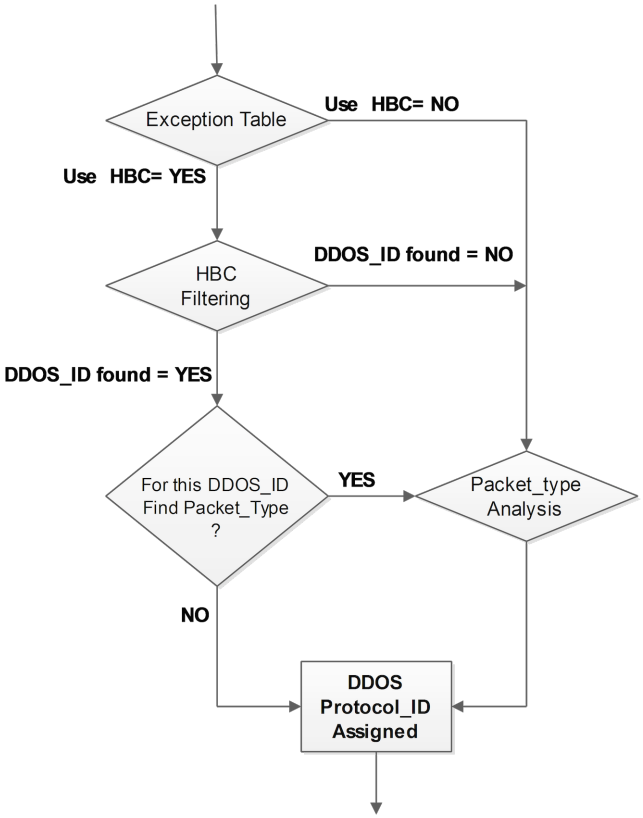



Figure 3.6 Packet Identification Logic

If an exception doesn't match any terms of the HBC filter or the HBC filter has been bypassed, the LU chip performs a second analysis to identify the protocol or type of the packet. Finally, if the LU chip is not able to identify the packet, it will classify it as *unclassified* or *IPv4-Unclassified* – categories which also have a dedicated hierarchical policer.

Figure 3.6 illustrates the packet type identification logic. In the case of ICMP packets, the LU chip does not perform subsequent packet identification: just HBC is enough. In other words, ICMP packets follow the arrow chain on the left of the diagram.

Let's continue our tangent from the path of the echo request packet, and analyze, in detail, the packet identification chain and the DDOS protection feature.

First of all, let's look at the HBC filter. Call the following PFE command (46137345 is the FWF index previously retrieved for this HBC filter):

```
NPC11(R2 vty)# show filter index 46137345  program
Filter index = 46137345
Optimization flag: 0x0
Filter notify host id = 0
Filter properties: None
Filter state = CONSISTENT
term HOSTBOUND_IGMP_TERM
term priority 0
   payload-protocol
       2
```

```
   then
      accept
      ddos proto 69
term HOSTBOUND_OSPF_TERM
term priority 0
   payload-protocol
       89
   then
      accept
      ddos proto 70
term HOSTBOUND_RSVP_TERM
[…]
```

NOTE The preceding output has been truncated.

This firewall filter is just a set of terms that try to match a given IPv4 protocol. When an entry matches a given protocol, the action *then* assigns it a *ddos proto* value (the DDOS_ID). The following table summarizes all the protocols that the HOST-BOUND_IPv4_FILTER filter is able to match, as per the Junos release installed in the DUT (R2).

Table 3.1 The HBC Filter Terms

| | Match IPv4 Proto | Match Other Fields | Assign Ddos Proto |
|---|---|---|---|
| HOSTBOUND_IGMP_TERM | 2 | NA | 69 |
| HOSTBOUND_OSPF_TERM | 89 | NA | 70 |
| HOSTBOUND_RSVP_TERM | 46 | NA | 71 |
| HOSTBOUND_PIM_TERM | 13 | NA | 72 |
| HOSTBOUND_DHCP_TERM | 17 | DST Port 67-68 | 24 |
| HOSTBOUND_RIP_TERM | 17 | DST Port 520-521 | 73 |
| HOSTBOUND_PTP_TERM | 17 | DST Port 319-320 | 74 |
| HOSTBOUND_BFD_TERM1 | 17 | DST Port 3784-3785 | 75 |
| HOSTBOUND_BFD_TERM2 | 17 | DST Port 4784 | 75 |
| HOSTBOUND_LMP_TERM | 17 | DST Port 701 | 76 |
| HOSTBOUND_ANCP_TERM | 6 | DST Port 6068 | 85 |
| HOSTBOUND_LDP_TERM1 | 6 | DST Port 646 | 77 |
| HOSTBOUND_LDP_TERM2 | 6 | SRC Port 646 | 77 |
| HOSTBOUND_LDP_TERM3 | 17 | DST Port 646 | 77 |
| HOSTBOUND_LDP_TERM4 | 17 | SRC Port 646 | 77 |
| HOSTBOUND_MSDP_TERM1 | 6 | DST Port 639 | 78 |
| HOSTBOUND_MSDP_TERM2 | 6 | SRC Port 639 | 78 |
| HOSTBOUND_BGP_TERM1 | 6 | DST Port 179 | 79 |
| HOSTBOUND_BGP_TERM2 | 6 | SRC Port 179 | 79 |

| | | | |
|---|---|---|---|
| HOSTBOUND_VRRP_TERM | 112 | IP DST 224.0.0.18/32 | 80 |
| HOSTBOUND_TELNET_TERM1 | 6 | DST Port 23 | 81 |
| HOSTBOUND_TELNET_TERM2 | 6 | SRC Port 23 | 81 |
| HOSTBOUND_FTP_TERM1 | 6 | DST Port 20-21 | 82 |
| HOSTBOUND_FTP_TERM2 | 6 | SRC Port 20-21 | 82 |
| HOSTBOUND_SSH_TERM1 | 6 | DST Port 22 | 83 |
| HOSTBOUND_SSH_TERM2 | 6 | SRC Port 22 | 83 |
| HOSTBOUND_SNMP_TERM1 | 17 | DST Port 161 | 84 |
| HOSTBOUND_SNMP_TERM2 | 17 | SRC Port 161 | 84 |
| HOSTBOUND_DTCP_TERM | 17 | DST Port 652 | 147 |
| HOSTBOUND_RADIUS_TERM_SERVER | 17 | DST Port 1812 | 150 |
| HOSTBOUND_RADIUS_TERM_ACCOUNT | 17 | DST Port 1813 | 151 |
| HOSTBOUND_RADIUS_TERM_AUTH | 17 | DST Port 3799 | 152 |
| HOSTBOUND_NTP_TERM | 17 | DST Port 123 & IP DST 224.0.0.1 | 153 |
| HOSTBOUND_TACACS_TERM | 17 | DST Port 49 | 154 |
| HOSTBOUND_DNS_TERM1 | 6 | DST Port 53 | 155 |
| HOSTBOUND_DNS_TERM2 | 17 | DST Port 53 | 155 |
| HOSTBOUND_DIAMETER_TERM1 | 6 | DST Port 3868 | 156 |
| HOSTBOUND_DIAMETER_TERM2 | 132 | DST Port 3868 | 156 |
| HOSTBOUND_L2TP_TERM | 17 | DST Port 171 | 161 |
| HOSTBOUND_GRE_TERM | 47 | NA | 162 |
| **HOSTBOUND_ICMP_TERM** | **1** | **NA** | **68** |
| OSTBOUND_TCP_FLAGS_TERM_INITIAL | 6 | Check TCP flags | 145 |
| HOSTBOUND_TCP_FLAGS_TERM_ESTAB | 6 | Check TCP flags | 146 |
| HOSTBOUND_TCP_FLAGS_TERM_UNCLS | 6 | Check TCP flags | 144 |
| HOSTBOUND_IP_FRAG_TERM_FIRST | 6 | Check TCP flags + Frag Offset | 159 |
| HOSTBOUND_IP_FRAG_TERM_TRAIL | 6 | Check TCP flags + Frag Offset | 160 |
| HOSTBOUND_AMT_TERM1 | 17 | DSP Port 2268 | 196 |
| HOSTBOUND_AMT_TERM2 | 17 | SRC Port 2268 | 196 |
| HOSTBOUND_IPV4_DEFAULT_TERM | ANY | NA | NA |

The ping traffic (IPv4 Protocol field = 1) is assigned to the DDOS_ID type 68, but remember this DDOS ID is not the final Protocol ID. To see the Protocol ID use the following PFE command:

```
NPC0(R2 vty)# show ddos asic punt-proto-maps
PUNT exceptions directly mapped to DDOS proto:
code PUNT name                      group proto         idx q# bwidth  burst
---- --------------------      --------- ------       ---- -- ------ ------
   1 PUNT_TTL                         ttl aggregate    3c00  5   2000  10000
   3 PUNT_REDIRECT                redirect aggregate    3e00  0   2000  10000
   5 PUNT_FAB_OUT_PROBE_PKT     fab-probe aggregate    5700  0  20000  20000
   7 PUNT_MAC_FWD_TYPE_HOST      mac-host aggregate    4100  2  20000  20000

[...]

PUNT exceptions that go through HBC. See following parsed proto
code PUNT name
---- -------------
   2 PUNT_OPTIONS     |
   4 PUNT_CONTROL     |
   6 PUNT_HOST_COPY   |
  11 PUNT_MLP         |---------------+
  32 PUNT_PROTOCOL    |               |
  34 PUNT_RECEIVE     |               |
  54 PUNT_SEND_TO_HOST_FW |           |
                                      |
         ------------------------------------------------------------------
         type   subtype            group proto         idx q# bwidth  burst
         ------ ----------     ---------- ----------   ---- -- ------ ------
[...]
         filter ipv4                icmp aggregate     900  0    500     50
[...]
```

As you can see, exception type 32 (remember PUNT code 32, also known as a *host route* exception) triggers HBC lookup and finally the Protocol ID for ICMP protocol (DDOS_ID 68) is: 0x900 (the idx column is the Protocol ID).

Now let's have a look at the DDOS policers. These are fully configurable via the Junos CLI. A policer refers to a specific protocol (Protocol ID at PFE level) and is made up of two parameters:

- The Bandwidth in PPS
- The burst size in number of packets

```
system {
   ddos-protection {
      protocols {
         <proto> {
           aggregate {
               bandwidth <pps>;
               burst <packets>;
            }
         <packet-type> {
               bandwidth <pps>;
               burst <packets>;
            }
         }
      }
   }
}
```

NOTE Variable parameters are enclosed between < >.

Figure 3.7 summarizes the DDOS protection concepts.

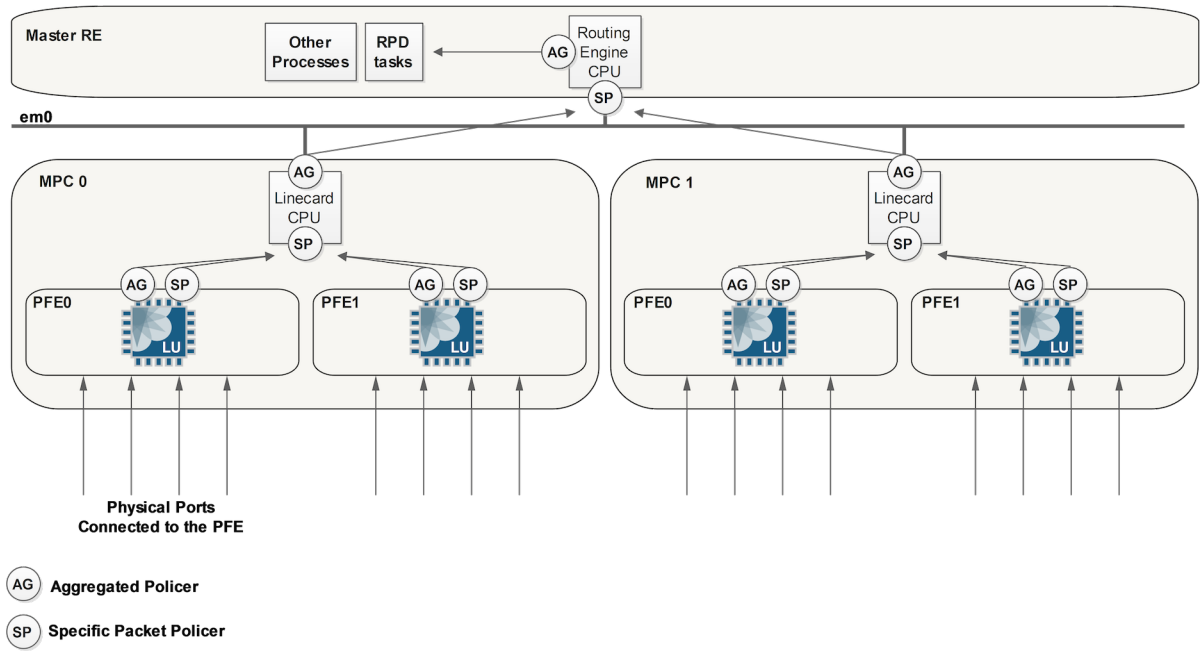



Figure 3.7 Hierarchical Anti-DDOS Policers

The main idea behind this model is hierarchical protection. Imagine there is a burst of incoming control or exception traffic arriving at (MPC0, PFE0) from one of its connected ports. The LU0 policers limit the amount of control or exception traffic that is sent up to the linecard CPU. So if there is a small amount of control or exception traffic arriving at (MPC0, PFE1), it still has a significant chance of being processed and not dropped.

Likewise, if there is an incoming storm of control or exception traffic arriving at both (MPC0, PFE0) and (MPC0, PFE1), the MPC0 CPU policer limits the traffic going up to the routing engine, thus giving a chance to the legitimate traffic being punted by MPC1 to be successfully processed by the RE.

These policers are especially effective because they are protocol-specific. For any given known protocol there is at least one policer, the global policer for the protocol (for all packet types). It is called the *aggregated policer*. An instance of this aggregated policer might be applied to one or more of the three host levels: the LU chip (HW policer), the µKernel's CPU (SW Policer), and the RE's CPU (SW Policer). When you configure the available bandwidth or burst of a given aggregated policer, the value is automatically assigned to the three instances of the policer. For some specific protocols, a per-packet-type policer could be available at some levels. *Packet type* policers apply to a sub-set of a given protocol. Bandwidth/burst values of a packet type policer are also automatically assigned to the three instances of the policer.

## A More Complex Example: DHCP

The atypical example is DHCP. DHCP is a protocol that uses several types of messages: DISCOVER, REQUEST, OFFER. Each has a dedicated policer, and an instance of this dedicated policer may be applied at the µKernel and RE's CPU. Globally, the DHCP protocol also has a policer: the aggregated one, and this one is applied to all levels.

There is a strict priority between per-packet-type policers: higher priority will always be served before a lower one. Figure 3.8 shows where aggregated (AG) and specific policers (SP) are applied to the DHCP protocol.

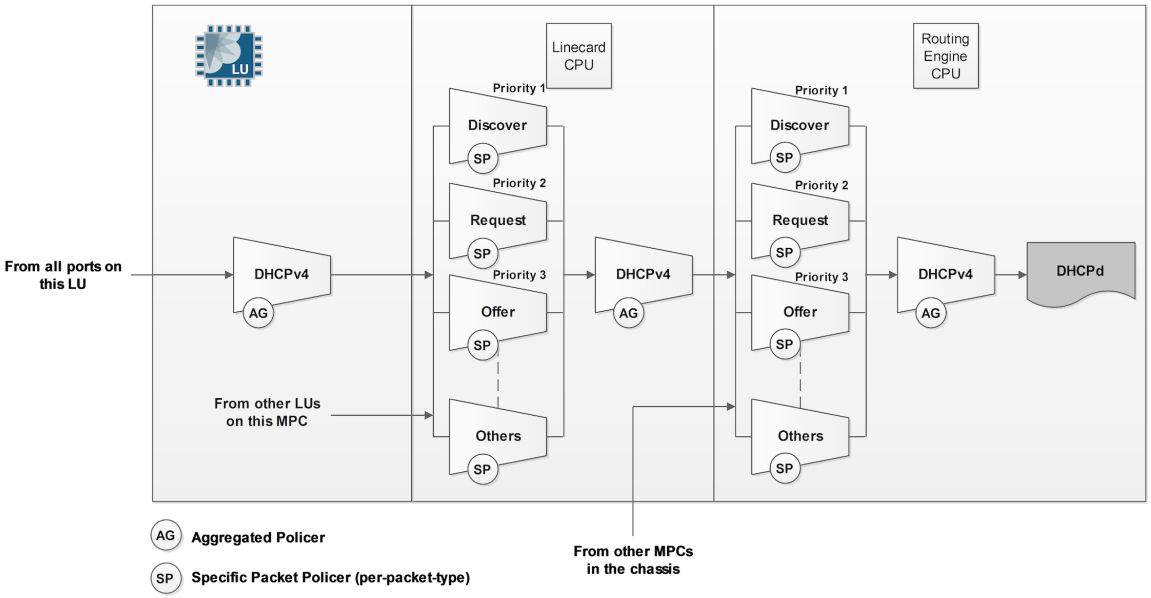



Figure 3.8 Packet-Type and Aggregated Policers Example: DHCP

Figure 3.8 is the DHCP case and it's not the same for other protocols. Indeed, for DHCP there is no SP policer at the LU chip level but for other protocols you might have one. Depending on the protocol, the AG or SP policer might be available at one or more levels.

Now the questions of our tangent are: how do you know where an AG policer is applied for a given protocol, and are there per-packet type policers (SP) available? If yes, where they are applied? Finally you also want to know what the default configured value is for each of these policers.

The first thing to keep in mind is this; if the AG or SP policers are configured at the µKernel Level, the same will be done at the RE level. In other words, if one aggregated policer plus two packet type policers are applied for a given protocol at the µKernel level, you will have the same set of policers at the RE level: meaning one AG plus two SPs.

In order to answer the questions above you can use CLI and PFE commands. For example, policer parameters can be retrieved by using the following CLI command. (Let's use DHCPv4 for a complex example, and then look at the simpler ICMP case):

```
user@R2-re0> show ddos-protection protocols dhcpv4 parameters brief
Packet types: 19, Modified: 0
* = User configured value
Protocol    Packet      Bandwidth Burst  Priority Recover   Policer  Bypass FPC
group       type        (pps)     (pkts)         time(sec) enabled  aggr.  mod
dhcpv4      aggregate   5000      5000   --       300       yes      --     no
dhcpv4      unclass..   300       150    Low      300       yes       no    no
dhcpv4      discover    500       500    Low      300       yes       no    no
dhcpv4      offer       1000      1000   Low      300       yes       no    no
dhcpv4      request     1000      1000   Medium   300       yes       no    no
dhcpv4      decline     500       500    Low      300       yes       no    no
dhcpv4      ack         500       500    Medium   300       yes       no    no
dhcpv4      nak         500       500    Low      300       yes       no    no
dhcpv4      release     2000      2000   High     300       yes       no    no
dhcpv4      inform      500       500    Low      300       yes       no    no
dhcpv4      renew       2000      2000   High     300       yes       no    no
dhcpv4      forcerenew  2000      2000   High     300       yes       no    no
dhcpv4      leasequery  2000      2000   High     300       yes       no    no
dhcpv4      leaseuna..  2000      2000   High     300       yes       no    no
dhcpv4      leaseunk..  2000      2000   High     300       yes       no    no
dhcpv4      leaseact..  2000      2000   High     300       yes       no    no
dhcpv4      bootp       300       300    Low      300       yes       no    no
dhcpv4      no-msgtype  1000      1000   Low      300       yes       no    no
dhcpv4      bad-pack..  0         0      Low      300       yes       no    no
```

As you can see, this command gives you the value of the aggregated policer for a given protocol (DHCP), and, if available, the value of each per-packet type policers. Each per-packet-type policer also has a priority associated with it. In this DHCP case, for example, the DHCP discover packets are served after DHCP Request packets.

## Back to the Basics: ICMP

Let's have a look at the DDOS policer configuration for ICMP:

```
user@R2-re0> show ddos-protection protocols icmp parameters brief
Packet types: 1, Modified: 0
* = User configured value
Protocol    Packet      Bandwidth Burst  Priority Recover   Policer  Bypass FPC
group       type        (pps)     (pkts)         time(sec) enabled  aggr.  mod
icmp        aggregate   20000     20000  --       300       yes      --     no
```

Indeed, you can see it's much simpler than DHCP. ICMP is managed globally, so there is no sub-type detection triggered by the LU chip (during packet identification phase) for ICMP protocol. As you can see, ICMP packets are rate-limited to 20000 pps. But the second question was, how many instances of this policer do you have and where are they applied?

To precisely answer this question you can use a PFE command that has a double advantage – it tells you where a policer is applied and also provides the real time associated statistics (packets passed / dropped / rate). These statistics are for the MPC that you executed the command on. Let's try to use the command on R2 on MPC 11:

```
NPC11(R2 vty)# show ddos policer icmp stats terse
DDOS Policer Statistics:
arrival   pass  # of
 idx prot     group      proto on     loc       pass    drop    rate   rate flows
 ---  ---  --------  --------- --  ------   --------   -----  ------ ------ -----
  68  900      icmp  aggregate  Y   UKERN     283248       0      10     10     0
                                   PFE-0     283248       0      10     10     0
                                   PFE-1          0       0       0      0     0
                                   PFE-2          0       0       0      0     0
                                   PFE-3          0       0       0      0     0
```

You can deduce from this output that the ICMP AG policer is applied at three levels. The RE level is not shown but it's implied since the existence of a policer at a low level (µKernel's CPU) implies its existence at all the higher levels. Moreover you can see (again): the DDOS ID of ICMP (which is 68 and is assigned by the HBC filter); and the Protocol ID associated to the ICMP aggregated policer, which is 0x900. This Protocol ID will be carried with the Parcel or the packet all the way to reach the Host. So, you can see:

- The LU chip level (PFE-0): ICMP is rate-limited to 20Kpps, meaning 20Kpps in total after adding up the incoming ICMP traffic at all the (four in case of the 16x10GE MPC) ports connected to this PFE.
- The µKernel level of MPC in slot 11: ICMP is rate-limited to 20Kpps, meaning 20Kpps in total after adding up the incoming ICMP traffic at all the (four) PFEs of this MPC (for example, for the sixteen ports of the MPC 16x10GE).
- The RE Level: ICMP is rate-limited to 20Kpps, meaning 20 Kpps in total for all the MPCs on this chassis altogether. Remember even if it is not explicitly displayed, the configuration of the µKernel level gives you the configuration of the RE level *(but here you cannot see it, because you are at MPC level)*.

Indeed, to retrieve global stats at the RE level, you have to use the following CLI command:

```
user@R2> show ddos-protection protocols icmp statistics terse
Packet types: 1, Received traffic: 1, Currently violated: 0

Protocol    Packet      Received        Dropped        Rate     Violation State
group       type        (packets)       (packets)      (pps)    counts
icmp        aggregate   710382          0              20       0         ok
```

Why do you see twenty pps at the RE level? Remember there are two pings that target the R2 router: ten pps from R1 (connected to MPC in slot 11) and ten pps from R3 (connected to MPC in slot 0). Removing the "terse" option of the previous command shows:

```
user@R2> show ddos-protection protocols icmp statistics
Packet types: 1, Received traffic: 1, Currently violated: 0

Protocol Group: ICMP

  Packet type: aggregate
    System-wide information:
      Aggregate bandwidth is never violated
      Received:  717710              Arrival rate:     20 pps
      Dropped:   0                   Max arrival rate: 20 pps
    Routing Engine information:
      Aggregate policer is never violated
      Received:  717884              Arrival rate:     20 pps
      Dropped:   0                   Max arrival rate: 20 pps
        Dropped by individual policers: 0
    FPC slot 0 information:
      Aggregate policer is never violated
      Received:  4462                Arrival rate:     10 pps
      Dropped:   0                   Max arrival rate: 10 pps
        Dropped by individual policers: 0
        Dropped by flow suppression:    0
    FPC slot 11 information:
      Aggregate policer is never violated
      Received:  713248              Arrival rate:     10 pps
      Dropped:   0                   Max arrival rate: 10 pps
        Dropped by individual policers: 0
        Dropped by flow suppression:    0
```

The per FPC statistics are actually the µKernel's statistics for each MPC. You cannot see per PFE (per LU) statistics via this CLI command. This information is only available with the previous PFE command: show ddos policer icmp stats terse.

So after this quick overview of ICMP DDoS statistics at the RE level, let's move back to the PFE level. Your ICMP echo request is still inside the ingress LU chip. The ten pps ICMP stream passed without any problem the lo0.0 firewall and then the aggregated HW policer of the LU chip. It's time for the ping to go out of the LU chip but before that happens, the LU chip must assign a hardware queue number to reach the µKernel's CPU.

## Queue Assignment

Remember, the LU chip only assigns the queue number and does not perform queuing and scheduling. These tasks are performed by the MQ chip. The queue assignment depends on the exception type, and for some exceptions also the packet type and more. To find a given exception packet's assigned queue, you can use a PFE command (used previously).

REMEMBER The ping to host stream is considered as Host Route Exception (remember PUNT(32) code).

The "q#" column gives you the associated HW queue:

```
NPC0(R2 vty)# show ddos asic punt-proto-maps
PUNT exceptions directly mapped to DDOS proto:
code PUNT name                      group proto       idx q# bwidth  burst
---- --------------------      --------- ------     ---- -- ------ ------
   1 PUNT_TTL                         ttl aggregate  3c00  5   2000  10000
   3 PUNT_REDIRECT                redirect aggregate  3e00  0   2000  10000
   5 PUNT_FAB_OUT_PROBE_PKT     fab-probe aggregate  5700  0  20000  20000
   7 PUNT_MAC_FWD_TYPE_HOST      mac-host aggregate  4100  2  20000  20000
   8 PUNT_TUNNEL_FRAGMENT        tun-frag aggregate  4200  0   2000  10000
  11 PUNT_MLP                         mlp packets    3802  2   2000  10000
  12 PUNT_IGMP_SNOOP           mcast-snoop igmp      4302  4  20000  20000
  13 PUNT_VC_TTL_ERROR           vchassis vc-ttl-err  805  2   4000  10000
  14 PUNT_L2PT_ERROR                l2pt aggregate  5a00  2  20000  20000
  18 PUNT_PIM_SNOOP            mcast-snoop pim       4303  4  20000  20000
  35 PUNT_AUTOSENSE               dynvlan aggregate   300  2   1000    500
  38 PUNT_SERVICES               services BSDT       4403  0  20000  20000
  39 PUNT_DEMUXAUTOSENSE        demuxauto aggregate  4500  0   2000  10000
  40 PUNT_REJECT                   reject aggregate  4600  6   2000  10000
  41 PUNT_SAMPLE_SYSLOG            sample syslog     5602  7   1000   1000
  42 PUNT_SAMPLE_HOST              sample host       5603  7  12000  12000
  43 PUNT_SAMPLE_PFE               sample pfe        5604  7   1000   1000
  44 PUNT_SAMPLE_TAP               sample tap        5605  7   1000   1000
  45 PUNT_PPPOE_PADI                pppoe padi        502  2    500    500
  46 PUNT_PPPOE_PADR                pppoe padr        504  3    500    500
  47 PUNT_PPPOE_PADT                pppoe padt        506  3   1000   1000
  48 PUNT_PPP_LCP                     ppp lcp         402  2  12000  12000
  49 PUNT_PPP_AUTH                    ppp auth        403  3   2000   2000
  50 PUNT_PPP_IPV4CP                  ppp ipcp        404  3   2000   2000
  51 PUNT_PPP_IPV6CP                  ppp ipv6cp      405  3   2000   2000
  52 PUNT_PPP_MPLSCP                  ppp mplscp      406  3   2000   2000
  53 PUNT_PPP_UNCLASSIFIED_CP         ppp unclass     401  2   1000    500
  55 PUNT_VC_HI                  vchassis control-hi  802  3  10000   5000
  56 PUNT_VC_LO                  vchassis control-lo  803  2   8000   3000
  57 PUNT_PPP_ISIS                    ppp isis        407  3   2000   2000
```

```
  58 PUNT_KEEPALIVE            keepalive aggregate  5b00  3  20000  20000
  59 PUNT_SEND_TO_HOST_FW_INLINE_SVCS inline-svcs aggregate   5d00  2  20000  20000
  60 PUNT_PPP_LCP_ECHO_REQ           ppp echo-req    408  2  12000  12000
  61 PUNT_INLINE_KA           inline-ka aggregate  5c00  3  20000  20000
  63 PUNT_PPP_LCP_ECHO_REP           ppp echo-rep    409  2  12000  12000
  64 PUNT_MLPPP_LCP                  ppp mlppp-lcp   40a  2  12000  12000
  65 PUNT_MLFR_CONTROL        frame-relay frf15     5e02  2  12000  12000
  66 PUNT_MFR_CONTROL         frame-relay frf16     5e03  2  12000  12000
  68 PUNT_REJECT_V6             rejectv6 aggregate  5900  6   2000  10000
  70 PUNT_SEND_TO_HOST_SVCS     services packet     4402  1  20000  20000
  71 PUNT_SAMPLE_SFLOW            sample sflow      5606  7   1000   1000

PUNT exceptions that go through HBC. See following parsed proto
code PUNT name
---- -------------
   2 PUNT_OPTIONS     |
   4 PUNT_CONTROL     |
   6 PUNT_HOST_COPY   |
  11 PUNT_MLP         |---------------+
  32 PUNT_PROTOCOL    |               |
  34 PUNT_RECEIVE     |               |
  54 PUNT_SEND_TO_HOST_FW |           |
                                      |
        ---------------------------------------------------------------------
        type   subtype            group proto       idx q# bwidth  burst
        ------ ----------    ---------- ---------- ---- -- ------ ------
        contrl LACP                lacp aggregate  2c00  3  20000  20000
        contrl STP                  stp aggregate  2d00  3  20000  20000
        contrl ESMC                esmc aggregate  2e00  3  20000  20000
        contrl OAM_LFM          oam-lfm aggregate  2f00  3  20000  20000
        contrl EOAM                eoam aggregate  3000  3  20000  20000
        contrl LLDP                lldp aggregate  3100  3  20000  20000
        contrl MVRP                mvrp aggregate  3200  3  20000  20000
        contrl PMVRP              pmvrp aggregate  3300  3  20000  20000
        contrl ARP                  arp aggregate  3400  2  20000  20000
        contrl PVSTP              pvstp aggregate  3500  3  20000  20000
        contrl ISIS                isis aggregate  3600  1  20000  20000
        contrl POS                  pos aggregate  3700  3  20000  20000
        contrl MLP                  mlp packets    3802  2   2000  10000
        contrl JFM                  jfm aggregate  3900  3  20000  20000
        contrl ATM                  atm aggregate  3a00  3  20000  20000
        contrl PFE_ALIVE      pfe-alive aggregate  3b00  3  20000  20000
        filter ipv4              dhcpv4 aggregate   600  0    500     50
        filter ipv6              dhcpv6 aggregate   700  0   5000   5000
        filter ipv4                icmp aggregate   900  0    500     50
        filter ipv4                igmp aggregate   a00  1  20000  20000
        filter ipv4                ospf aggregate   b00  1  20000  20000
        filter ipv4                rsvp aggregate   c00  1  20000  20000
        filter ipv4                 pim aggregate   d00  1   8000  16000
        filter ipv4                 rip aggregate   e00  1  20000  20000
        filter ipv4                 ptp aggregate   f00  1  20000  20000
        filter ipv4                 bfd aggregate  1000  1  20000  20000
        filter ipv4                 lmp aggregate  1100  1  20000  20000
        filter ipv4                 ldp aggregate  1200  1    500    500
        filter ipv4                msdp aggregate  1300  1  20000  20000
        filter ipv4                 bgp aggregate  1400  0   1000   1000
        filter ipv4                vrrp aggregate  1500  1    500    500
        filter ipv4              telnet aggregate  1600  0  20000  20000
        filter ipv4                 ftp aggregate  1700  0  20000  20000
        filter ipv4                 ssh aggregate  1800  0  20000  20000
        filter ipv4                snmp aggregate  1900  0  20000  20000
        filter ipv4                ancp aggregate  1a00  1  20000  20000
        filter ipv6              igmpv6 aggregate  1b00  1  20000  20000
```

```
         filter ipv6               egpv6 aggregate  1c00  1  20000  20000
         filter ipv6              rsvpv6 aggregate  1d00  1  20000  20000
         filter ipv6            igmpv4v6 aggregate  1e00  1  20000  20000
         filter ipv6               ripv6 aggregate  1f00  1  20000  20000
         filter ipv6               bfdv6 aggregate  2000  1  20000  20000
         filter ipv6               lmpv6 aggregate  2100  1  20000  20000
         filter ipv6               ldpv6 aggregate  2200  1  20000  20000
         filter ipv6              msdpv6 aggregate  2300  1  20000  20000
         filter ipv6               bgpv6 aggregate  2400  0  20000  20000
         filter ipv6              vrrpv6 aggregate  2500  1  20000  20000
         filter ipv6            telnetv6 aggregate  2600  0  20000  20000
         filter ipv6               ftpv6 aggregate  2700  0  20000  20000
         filter ipv6               sshv6 aggregate  2800  0  20000  20000
         filter ipv6              snmpv6 aggregate  2900  0  20000  20000
         filter ipv6              ancpv6 aggregate  2a00  1  20000  20000
         filter ipv6            ospfv3v6 aggregate  2b00  1  20000  20000
         filter ipv4           tcp-flags unclass..  4801  0  20000  20000
         filter ipv4           tcp-flags initial    4802  0  20000  20000
         filter ipv4           tcp-flags establish  4803  0  20000  20000
         filter ipv4                dtcp aggregate  4900  0  20000  20000
         filter ipv4              radius server     4a02  0  20000  20000
         filter ipv4              radius account..  4a03  0  20000  20000
         filter ipv4              radius auth..     4a04  0  20000  20000
         filter ipv4                 ntp aggregate  4b00  0  20000  20000
         filter ipv4              tacacs aggregate  4c00  0  20000  20000
         filter ipv4                 dns aggregate  4d00  0  20000  20000
         filter ipv4             diameter aggregate  4e00  0  20000  20000
         filter ipv4              ip-frag first-frag 4f02  0  20000  20000
         filter ipv4              ip-frag trail-frag 4f03  0  20000  20000
         filter ipv4                 l2tp aggregate  5000  0  20000  20000
         filter ipv4                 gre aggregate  5100  0  20000  20000
         filter ipv4               ipsec aggregate  5200  0  20000  20000
         filter ipv6               pimv6 aggregate  5300  1   8000  16000
         filter ipv6              icmpv6 aggregate  5400  0  20000  20000
         filter ipv6               ndpv6 aggregate  5500  0  20000  20000
         filter ipv4               amtv4 aggregate  5f00  0  20000  20000
         filter ipv6               amtv6 aggregate  6000  0  20000  20000
         option rt-alert          ip-opt rt-alert   3d02  1  20000  20000
         option unclass           ip-opt unclass..  3d01  4  10000  10000

PUNT exceptions parsed by their own parsers
code PUNT name
---- -------------
 209 PUNT_RESOLVE     |
 209 PUNT_RESOLVE_V6  |---------------+
                                      |
         ------------------------------------------------------------------
                                resolve aggregate   100  0   5000  10000
                                resolve other       101  6   2000   2000
                                resolve ucast-v4    102  6   3000   5000
                                resolve mcast-v4    103  6   3000   5000
                                resolve ucast-v6    104  6   3000   5000
                                resolve mcast-v6    105  6   3000   5000

REJECT_FW exception mapped to DHCPv4/6 and filter-act. Only filter-act shown
   7 PUNT_REJECT_FW   |---------------+
                                      |
         ------------------------------------------------------------------
                              filter-act aggregate   200  0   5000  10000
                              filter-act other       201  6   2000   5000
                              filter-act filter-v4   202  6   2000   5000
                              filter-act filter-v6   203  6   2000   5000
```

And you can see that the ICMP echo request (Protocol ID 0x900) is assigned to queue number 0. And this last output ends our tangent on host inbound packet identification and DDOS protection

## Ping to the Host, Stage Three: From LU back to MQ/XM

Okay, the LU chip has performed packet lookup, packet filtering, packet identification, packet ddos rate-limiting, and packet queue assignment. Whew!

It's time for the Parcel to go back to the MQ/XM chip via the LU In block (LI). The Parcel is "rewritten" before moving back to the MQ/XM chip. Indeed, the LU chip adds some additional information in a Parcel header. The Parcel header includes the IFL index (incoming logical interface), the packet type (used by DDOS protection at the µKernel – the Protocol ID, which for ICMP is 0x900) and some other information.

NOTE This additional information is hidden from the MQ/XM chip – only the next L2M header will be understood by the MQ/XM chip.

After modifying the Parcel headers, the LU chip provides the MQ/XM chip with some information through an additional (L2M) header that gets prepended to the Parcel header. The L2M header conveys, among other things, the fact that it is a host-inbound packet, and the Host Queue Assigned by the LU (L2M conveys the absolute queue number that corresponds to the relative queue 0 - see below).

The MQ/XM chip reads the L2M header and learns that the packet is an exception packet for the host. The L2M header is then removed.

Then the MQ/XM chip internally conveys host traffic in a dedicated stream: the Host Stream always has the value 1151.

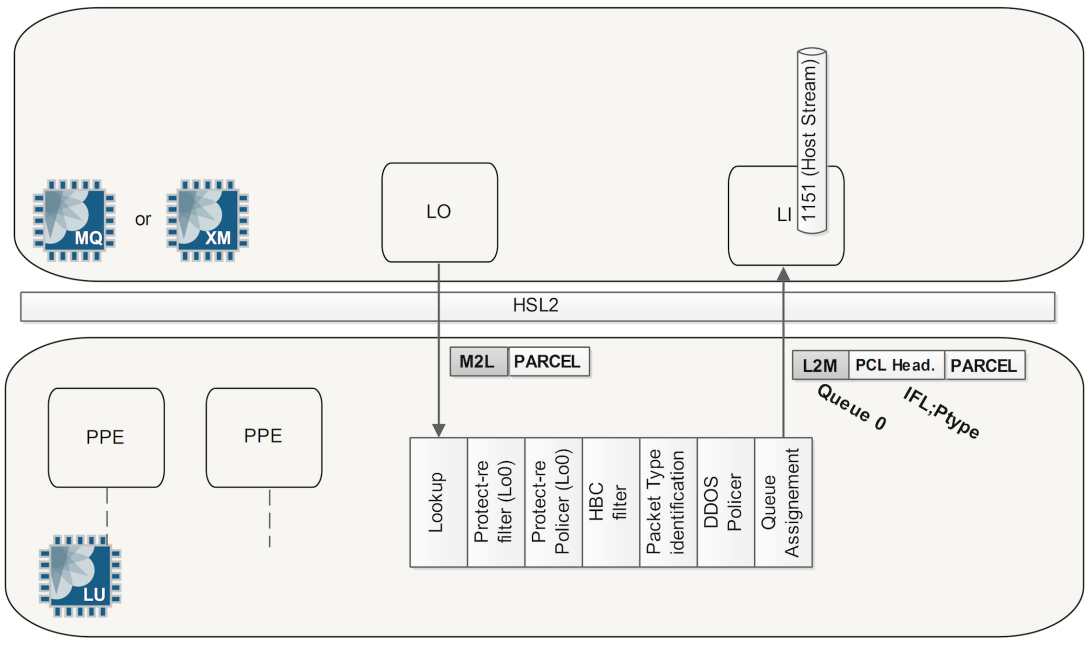



Figure 3.9 From LU Back to MQ/XM Chip - the L2M Header

## Ping to the Host, Stage Four: from MQ/XM to the Host

Host HW queues are managed by the MQ/XM chip as WAN queues (see the Appendix *WAN CoS* tangent). The host interface, as a WAN interface between the MQ or XM chip and the µKernel's CPU, has eight HW queues available.

NOTE The MQ/XM chip accesses the µKernel's CPU via DMA through a PCIe interface.

Since the host interface (stream 1151) is managed by the SCHED block has a WAN interface:

- The Q system 0 of the MQ or XM chip manages the delivery of host packets.
- There are eight queues (eight Q nodes) to carry traffic destined to the host.
- These queues are not configurable and each queue has a priority but not a guaranteed bandwidth. Table 3.2 lists the main role of each host queue:

Table 3.2 The Host Inbound Queues

| Queue Number | Main Role / Priority |
|---|---|
| Queue 0 | Layer 3 packets / low priority |
| Queue 1 | Layer 3 packets / high priority |
| Queue 2 | Layer 2 packets / low priority |
| Queue 3 | Layer 2 packets / high priority |
| Queue 4 | Queue for IP Options packets |
| Queue 5 | Queue for TTL packets |
| Queue 6 | Queue for Error Notification |
| Queue 7 | Queue for Sample / Syslog |

Host packets are delivered from the MQ or XM chip to the µKernel's CPU via the Trinity Offload Engine (TOE). TOE is an embedded component of the M chip that handles DMA to/from the µKernel's CPU. To notify the SCHED block that a specific queue of µKernel is overloaded, TOE sends a backpressure message to SCHED Block. Later we will see a case of congestion on one queue.

The host interface is not a real interface and you can't use the same commands to retrieve L1/L2/Q node information associated to the host interface. Nevertheless, another command is available to display the host interface information (Host Stream 1151).

NOTE This is the same command for both MQ and XM based cards.

```
NPC11(R2 vty)# show cos halp queue-resource-map 0  <<<< 0 means PFE_ID = 0
Platform type      : 3 (3)
FPC ID             : 0x997 (0x997)
Resource init      : 1
cChip type         : 1
Rich Q Chip present: 0
  Special stream count: 5
  ------------------------------------------------------------
  stream-id  L1-node  L2-node  base-Q-Node
  ------------------------------------------------------------
       1151      127      254         1016    <<<< Host Stream Base queue (Q) node
       1119       95      190          760
```

```
      1120        96       192          768
      1117        93       186          744
      1118        94       188          752
  ------------------------------------------------------------
```

For the MPC in slot 11 you can see that the CoS of the host interface is managed by the L1 node index 127 / L2 node index 254 and the eight Q nodes (1016 [the base Queue relative to queue 0] to 1023).

Calling the same command on the MPC in slot 0 gives you other values (this is an MPC4e card): the L1 node index 126, L2 node index 252, and the eight Q nodes (1008 [the base Queue] to 1015).

```
NPC0(R2 vty)# show cos halp queue-resource-map 0 <<<< 0 means PFE_ID = 0
Platform type      : 3 (3)
FPC ID             : 0xb4e (0xb4e)
Resource init      : 1
cChip type         : 2
Rich Q Chip present: 0
  Special stream count: 13
  ------------------------------------------------------------
  stream-id  L1-node  L2-node  base-Q-Node
  ------------------------------------------------------------
      1151      126      252         1008   <<<< HOST STREAM base queue (Q) node
```

Figure 3.10 illustrates the host stream CoS for MPC 0 and MPC 11.

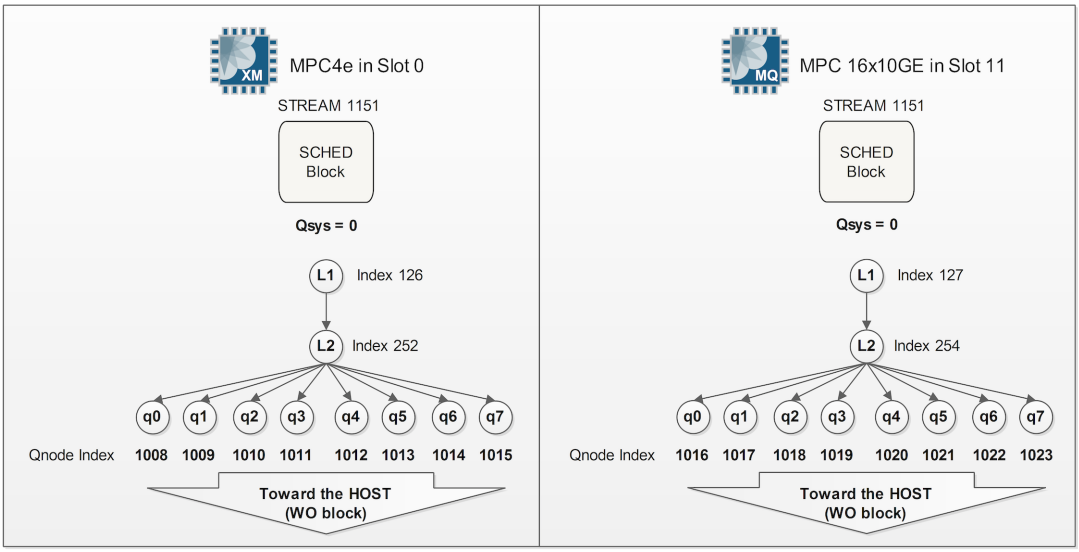



Figure 3.10 The Host Stream CoS Tree

The fact that the q-node numbers from one MPC to another are consecutive is simply a coincidence. The q-node numbers (which represent the absolute queue number) only have a local significance to each MPC.

The aim of the next PFE command sequence is to check the statistics of the host interface queues. Remember the ping packet has been assigned to the relative Queue 0 (Absolute Q node 1016).

REMEMBER MQ/XM chip has learned from the LU chip (L2M header) this information: Qsystem and Queue Number.

For MPC in slot 11, use this command:

```
NPC11(R2 vty)# show mqchip 0 dstat stats 0 1016 (<<<< 2nd 0 means Qsys 0)
  QSYS 0 QUEUE 1016 colormap 2 stats index 0:
                   Counter       Packets     Pkt Rate        Bytes    Byte Rate
  ----------------------- ------------- ------------ ------------ ------------
    Forwarded (NoRule)                0            0            0            0
    Forwarded (Rule)             447264           10     55013472         1230
    Color 0 Dropped (WRED)            0            0            0            0
    Color 0 Dropped (TAIL)            0            0            0            0
    Color 1 Dropped (WRED)            0            0            0            0
    Color 1 Dropped (TAIL)            0            0            0            0
    Color 2 Dropped (WRED)            0            0            0            0
    Color 2 Dropped (TAIL)            0            0            0            0
    Color 3 Dropped (WRED)            0            0            0            0
    Color 3 Dropped (TAIL)            0            0            0            0
    Dropped (Force)                   0            0            0            0
    Dropped (Error)                   0            0            0            0

  Queue inst depth     : 0
  Queue avg len (taql): 0
```

And for MPC in slot 0:

```
NPC0(R2 vty)# show xmchip 0 q-node stats 0 1008  (<<<< 2nd 0 means Qsys 0)
Queue statistics (Queue 1008)
-----------------------------
-------------------------------------------------------------------------------
Color  Outcome              Counter  Counter  Total                Rate
                            Index    Name
-------------------------------------------------------------------------------
All    Forwarded (No rule)  384      Packets  0                    0 pps
All    Forwarded (No rule)  384      Bytes    0                    0 bps
All    Forwarded (Rule)     385      Packets  44160                10 pps
All    Forwarded (Rule)     385      Bytes    5431680              9888 bps
All    Force drops          388      Packets  0                    0 pps
All    Force drops          388      Bytes    0                    0 bps
All    Error drops          389      Packets  0                    0 pps
All    Error drops          389      Bytes    0                    0 bps

0      WRED drops           386      Packets  0                    0 pps
0      WRED drops           386      Bytes    0                    0 bps
0      TAIL drops           387      Packets  0                    0 pps
0      TAIL drops           387      Bytes    0                    0 bps

1      WRED drops           390      Packets  0                    0 pps
1      WRED drops           390      Bytes    0                    0 bps
1      TAIL drops           391      Packets  0                    0 pps
1      TAIL drops           391      Bytes    0                    0 bps

2      WRED drops           392      Packets  0                    0 pps
2      WRED drops           392      Bytes    0                    0 bps
2      TAIL drops           393      Packets  0                    0 pps
2      TAIL drops           393      Bytes    0                    0 bps

3      WRED drops           394      Packets  0                    0 pps
3      WRED drops           394      Bytes    0                    0 bps
3      TAIL drops           395      Packets  0                    0 pps
3      TAIL drops           395      Bytes    0                    0 bps
-------------------------------------------------------------------------------
Drop structure
--------------
dcfg_taql                : 0x0 (Mantissa: 0, Shift: 0, Value: 0)
dcfg_instantaneous_depth : 0
```

You can see that MQ/XM sends the ICMP echo requests without any drops to the µKernel's CPU via the queue 0. Before they reach µKernel the packets still have to cross a last functional block of the MQ/XM chip: the Wan Output (WO) block. As mentioned, host interface is managed as a WAN interface. Therefore, the WO, in charge of managing WAN Output Streams, handles host packets of the Host Stream 1151 as well.

WO retrieves the entire echo request packet (with the specific Parcel header) from the packet buffer, reassembles the data units, and sends it to the µKernel's CPU via the TOE. If you need to check WO statistics you must configure Stream Accounting in advance. Indeed, as with the WI block, the PFE doesn't maintain per WAN Output Stream Statistics by default.

When it comes to activating WO statistics for a given stream, in this case the host stream 1151, there are some differences between the two models of MPCs and both are covered.

Let's first activate WO accounting for host stream on MPC in slot 11:

```
NPC11(R2 vty)# test mqchip 0 counter wo 0 1151  <<<< the second 0 is the counter 0 (WO supports 2
counters max)
```

Then display the WO statistics for MPC in slot 11:

```
NPC11(R2 vty)# show mqchip 0 counters output stream 1151
DSTAT phy_stream 1151 Queue Counters:
  Aggregated Queue Stats QSYS 0 Qs 1016..1023 Colors 0..3:
Counter          Packets     Pkt Rate       Bytes    Byte Rate
------------------------ -------------  --------- ------------
Forwarded       3658102         11   164530142       1264  <<< Aggregated Stat of the 8 queues
Dropped               0            0           0            0

WO Counters:
Stream  Mask Match       Packets  Pkt Rate     Bytes    Byte Rate   Cells
------------------------ -------  --------- --------- ------------  -----
0x07f 0x07f << Counter 0     128        10     14676         1202      20
```

Let's deactivate WO accounting for the host stream on the MPC in slot 11:

```
NPC11(R2 vty)# test mqchip 0 counter wo 0 default
```

Do the same for MPC in slot 0. Activate WO accounting for host stream:

```
NPC0(R2 vty)# test xmchip 0 wo stats stream 0 1151 <<<< the second 0 is the counter 0
```

And then display WO statistics for MPC in slot 0 :

```
NPC0(R2 vty)# show xmchip 0 phy-stream stats 1151 1 <<< 1 means Out Direction

Aggregated queue statistics
---------------------------
Queues: 1008..1015
------------------------------------------------------------------------
Color  Outcome             Counter  Total                Rate
                           Name
------------------------------------------------------------------------
All    Forwarded (No rule)  Packets  0                    0 pps
All    Forwarded (No rule)  Bytes    0                    0 bps
All    Forwarded (Rule)     Packets  3475669          11 pps<<<< Aggregated Stat of the 8 queues
All    Forwarded (Rule)     Bytes    126869375            10256 bps
[...]
------------------------------------------------------------------------
```

```
WO statistics (WAN Block 0)
---------------------------

Counter set 0

    Stream number mask       : 0x7f
    Stream number match      : 0x7f
    Transmitted packets      : 352 (10 pps)        <<<< stats counter 0
    Transmitted bytes        : 40448 (10008 bps)
    Transmitted cells        : 672 (20 cps)

Counter set 1

    Stream number mask       : 0x0
    Stream number match      : 0x0
    Transmitted packets      : 10402752 (22 pps)
    Transmitted bytes        : 373341245 (18312 bps)
    Transmitted cells        : 10672505 (42 cps)
```

Let's deactivate WO accounting for Host Stream on MPC in slot 0 to end our troubleshooting example:

```
NPC0(R2 vty)# test xmchip 0 wo stats default 0 0 <<< third 0 means WAN Group 0
```

Figure 3.11 depicts the entire path of the echo request to reach the µKernel's CPU.

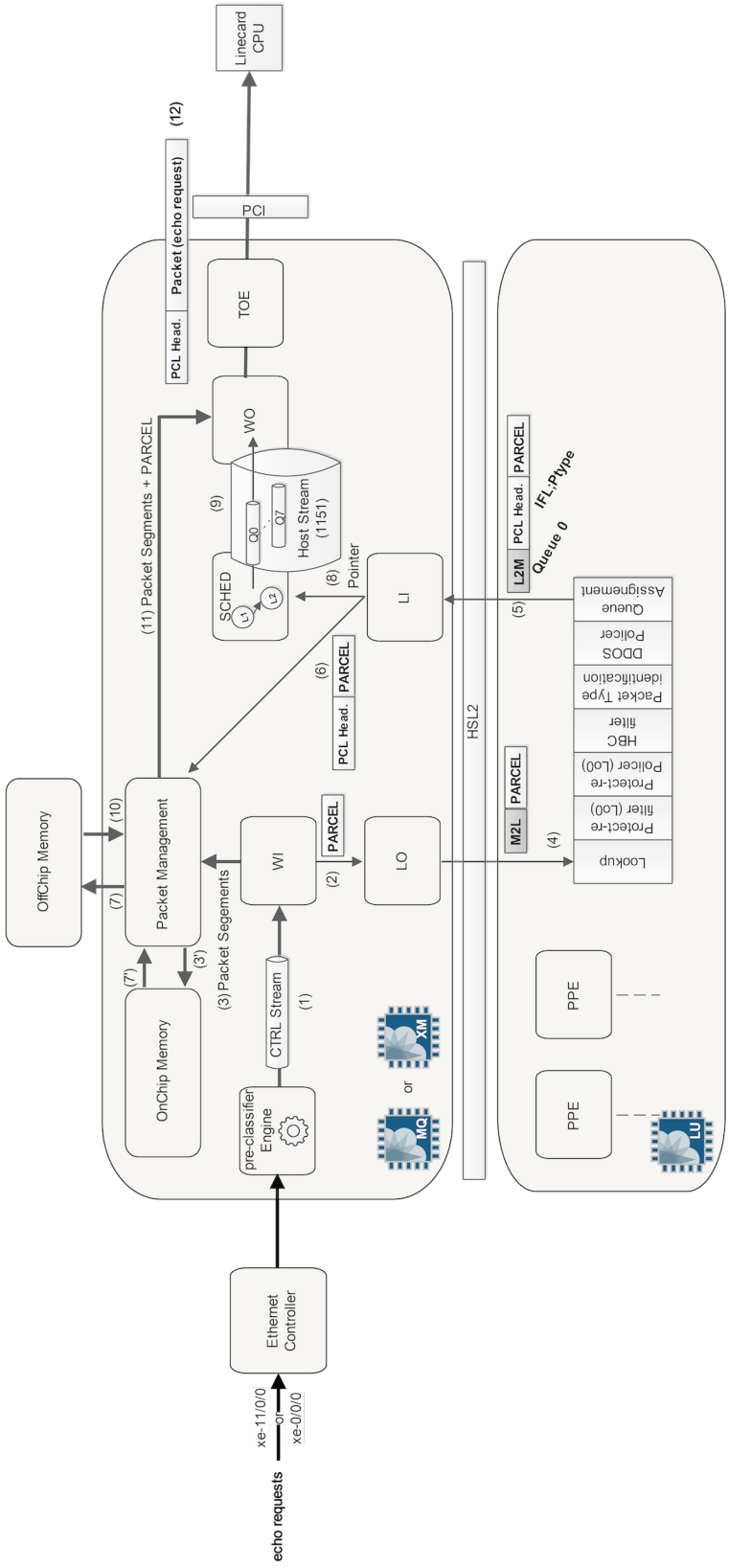



Figure 3.11 Global View of Incoming Ping Echo Request Inside the PFE

## Ping to the Host, Stage Five: at the Host

### In the Line Card CPU (microkernel)

The packet with its Parcel header (built by the LU chip) is received by the Line Card's CPU via DMA. The µKernel handles packets in software. It has its own micro operating system with several processes (threads), and we'll only highlight some of the threads that run on top of the µKernel CPU:

```
NPC11(R2 vty)# show threads
PID PR State     Name                   Stack Use  Time (Last/Max/Total) cpu
--- -- -------   --------------------   ---------  --------------------
  2 L  running   Idle                    320/2056   0/0/3331235606 ms 93%
 36 H  asleep    TTP Receive            840/4096   0/0/13412 ms  0%
 37 H  asleep    TTP Transmit          1136/4104   0/0/135540 ms  0%
 96 L  asleep    DDOS Policers         1976/4096   2/3/11302265 ms  0%
[...]
```

At this level the incoming ICMP echo request is managed by software queues (four queues – a merger of the eight HW queues coming from the MQ/XM chip). You should remember from our tangential discussion that DDOS policers are in charge of managing AG and SP DDOS policers. And the packet type (Protocol_ID) identified by the LU chip has followed the packet in the Parcel header. So the µKernel doesn't need to identify the type of exception. And the traffic stream is low enough for the DDOS protection at the µKernel to pass all packets, as you can check with the following CLI command (here the statistics are for MPC 11):

```
user@R2> show ddos-protection protocols icmp statistics | find "FPC slot 11"
    FPC slot 11 information:
      Aggregate policer is never violated
      Received:  945181              Arrival rate:     10 pps
      Dropped:   0                   Max arrival rate: 10 pps
        Dropped by individual policers: 0
        Dropped by flow suppression:    0
```

This CLI command shows you statistics on a per-MPC basis. Remember the Host Queues that lie between MQ/XM chip and the µKernel. Since there may be drops on these queues, not all the host/exception packets that pass out of the LU policers actually reach the µKernel.

TIP Check out this line card shell command: `show ddos policer stats icmp`.

### To the Routing Engine CPU

All's well here. Now, it's time to punt the packet to the RE via the internal Gigabit Ethernet interface (em0 on the MX). Again the packet is not passed directly to the RE. The µKernel and the RE use a proprietary protocol called Trivial Tunneling Protocol (TTP) to exchange external control traffic (see Figure 3.12). With external control traffic, we mean packets that are received from (or sent out to) the outside world. On the other hand, native control traffic (forwarding table, boot image statistics, etc.) are not encapsulated in TTP.

TTP allows some additional information to be conveyed in its header that may be used by some processes of the RE. TTP is a proprietary protocol that runs over IP in modern Junos releases.

NOTE Not all the control/exception packet types go up to the routing engine's CPU. Some are fully processed by the line card's CPU.

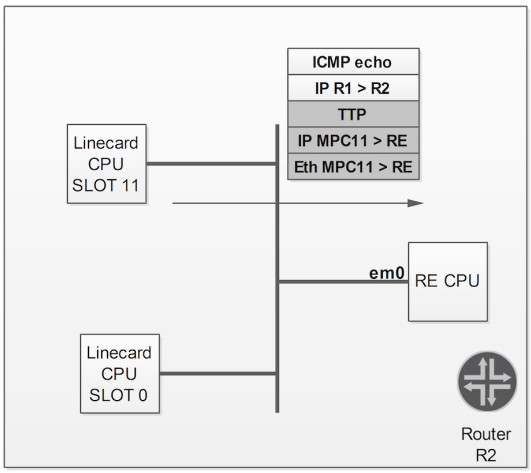



Figure 3.12 TTP Encapsulation Between the µKernel and the RE

Global TTP's statistics for a given MPC can be displayed by using this PFE command:

```
NPC11(R2 vty)# show ttp statistics
TTP Statistics:
[...]

TTP Transmit Statistics:
                  Queue 0     Queue 1     Queue 2     Queue 3
                ----------  ----------  ----------  ----------
 L2 Packets       6100386           0           0           0
 L3 Packets             0           0           0           0

TTP Receive Statistics:
                  Control        High      Medium         Low     Discard
                ----------  ----------  ----------  ----------  ----------
 L2 Packets             0           0           0           0           0
 L3 Packets             0           0    72630191           0           0
 Drops                  0           0           0           0           0
 Queue Drops            0           0           0           0           0
 Unknown                0           0           0           0           0
 Coalesce               0           0           0           0           0
 Coalesce Fail          0           0           0           0           0
[...]
```

Note that this command's output is quite ambiguous. Indeed, *TTP Transmit Statistics* means traffic received from the RE, then transmitted to the WAN. *Receive Statistics* means traffic received from the MQ/XM chip then transmitted to the RE. As you can see, the ICMP echo request is handled by the medium queue of the µKernel before it is sent to the RE. In this case, only the Layer 3 of the packet is sent. (The Layer 2 has been removed by the ingress LU chip.)

NOTE On the other hand, the host outbound packets sent by the RE are received by the microkernel CPU with the Layer 2 header (see Chapter 4) and put in queue 0 before they are sent to the MQ/XM chip.

Let's move on! An easy way to see the punted packet is to use the `monitor traffic interface` command on the em0 interface. If you try to use this command without any filter you will have some surprises. Indeed, a lot of native packets are exchanged between the RE and the MPCs. To limit the capture to a packet coming from a specific MPC, you must use the method explained next.

Each *management* interface of each MPC has a dedicated MAC and IP address. To retrieve the MPC MAC address you can use this hidden CLI command:

```
user@R2> show tnp addresses
   Name                TNPaddr   MAC address    IF     MTU E H R
master                   0x1 02:01:00:00:00:05 em0    1500 0 0 3
master                   0x1 02:01:01:00:00:05 em1    1500 0 1 3
re0                      0x4 02:00:00:00:00:04 em1    1500 2 0 3
re1                      0x5 02:01:00:00:00:05 em0    1500 0 0 3
re1                      0x5 02:01:01:00:00:05 em1    1500 0 1 3
backup                   0x6 02:00:00:00:00:04 em1    1500 2 0 3
fpc0                    0x10 02:00:00:00:00:10 em0    1500 4 0 3
fpc11                   0x1b 02:00:00:00:00:1b em0    1500 5 0 3
bcast             0xffffffff ff:ff:ff:ff:ff:ff em0    1500 0 0 3
bcast             0xffffffff ff:ff:ff:ff:ff:ff em1    1500 0 1 3
```

In this way, you can view the MAC address of the management interface of each non-passive component of the chassis. The IP address of each of them can be deduced from the last byte of the MAC address. Indeed, MPC's IP address starts with the prefix: 128.0.0/24. The last byte of the IP address is the same as the last byte of MAC address. So for MPC 11, the IP address is 128.0.0.27 (0x1b). For more information, execute the command `show arp vpn__juniper_private1__` .

Now, let's start to monitor the em0 interface and apply a filter to only capture traffic coming from the source MAC address 02:00:00:00:00:1b. The MPC11's MAC TTP protocol is not dissected by the Junos tcpdump. Only some information is displayed but using `print-hex print-ascii` can help you retrieve some other interesting information:

```
user@R2> monitor traffic interface em0 no-resolve layer2-headers matching "ether src host
02:00:00:00:00:1b" print-hex print-ascii size 1500
18:14:43.126104  In 02:00:00:00:00:1b > 02:01:00:00:00:05, ethertype IPv4 (0x0800), length 146:
128.0.0.27 > 128.0.0.1: TTP, type L3-rx (3), ifl_input 325, pri medium (3), length 92

0x0000   0201 0000 0005 0200 0000 001b 0800 4500        ..............E.
0x0010   0084 d8aa 0000 ff54 e25e 8000 001b 8000        .......T.^......
0x0020   0001 0303 0200 0000 0145 005c 0000 0000        .........E.\....
0x0030   0207 0000 8010 0004 0004 0900 0000 4500        <<< 0x0900 is there
0x0040   0054 a62a 0000 4001 545b ac10 1402 ac10        .T.*..@.T[......
0x0050   1401 0800 209c ccd0 00dd 538d 9646 0004        ..........S..F..
0x0060   8e62 4441 594f 4e45 2044 4159 4f4e 4520        .bDAYONE.DAYONE.
0x0070   4441 594f 4e45 2044 4159 4f4e 4520 4441        DAYONE.DAYONE.DA
0x0080   594f 4e45 2044 4159 4f4e 4520 4441 594f        YONE.DAYONE.DAYO
0x0090   4e45                                           NE
```

First of all you can see the incoming IFL (Interface Logical), which received the packet. IFL 325 is indeed the IFL allocated to the ae0.0 interface as shown here:

```
user@R2> show interfaces ae0.0 | match Index
  Logical interface ae0.0 (Index 325) (SNMP ifIndex 1019)
```

Next, if you deeply analyze the output, the packet you can retrieve in Hex-Mode is the DDOS Protocol_ID assigned by the LU chip. For ICMP's case, it is 0x900.

One more time, software policers handle host incoming packets at the RE level, and these policers are programmed by the jddosd process:

```
user@R2> show system processes | match jddos
81740  ??  I      7:24.67 /usr/sbin/jddosd -N
```

And the following command can be used to see the RE's DDOS protection statistics:

```
user@R2> show ddos-protection protocols icmp statistics terse
Packet types: 1, Received traffic: 1, Currently violated: 0

Protocol    Packet      Received        Dropped        Rate     Violation State
group       type        (packets)       (packets)      (pps)    counts
icmp        aggregate   1283280         0              20       0         ok
```

Remember, this CLI command gives you global statistics, meaning statistics with the RE point of view (the entire chassis). This is why the rate is 20 pps – it counts the two pings coming from MPC 11 and from MPC 0 (R1 and R3).

Finally, use the next CLI command to check traffic statistics and drops between the µKernel and the RE (except for the last line HW input drops, which counts drops between MQ/XM chip and µKernel):

```
user@R2> show pfe statistics traffic fpc 11
Packet Forwarding Engine traffic statistics:
    Input  packets:                   40                    10 pps
    Output packets:                   40                    10 pps
Packet Forwarding Engine local traffic statistics:
    Local packets input                : 40
    Local packets output               : 40
    Software input control plane drops  : 0 << Drop uKernel to RE
    Software input high drops           : 0 << Drop uKernel to RE
    Software input medium drops         : 0 << Drop uKernel to RE
    Software input low drops            : 0 << Drop uKernel to RE
    Software output drops               : 0 << Drop uKernel to PFE Stream 1151
    Hardware input drops                : 0 << Drop Stream 1151 of all PFE of the
                                           << MPC to µKernel
```

MORE? Execute show system statistics ttp to retrieve TTP global statistics from the RE perspective.

And let's check the global ICMP statistics:

```
user@R2> show system statistics icmp | match "Histo|echo"
         Output Histogram
                 685755 echo reply
                 676811 echo
         Input Histogram
                 615372 echo reply
                 685755 echo
```

Great! Our simple echo request has reached its destination: the *host* , without any issue!

This last command ends the host-inbound packet walkthrough. You witnessed a lot of concepts that will be taken for granted in the next sections of the book.

AND NOW? Technically, the deep dive of the host inbound path ends here. You can expand your knowledge about the host protection feature set in Appendix B. Or if you wish, just move on to Chapter 4 for a deep dive in the host outbound path. And don't forget to take a break if necessary!

# Chapter 4

## From the Host to the Outer World

Chapter 3 focused on the packet walkthough of incoming control traffic. This chapter examines the return path using two examples: ICMP packets generated by the Routing Engine, and LACP packets generated by the line card.

The host outbound path is quite different from the host inbound path. You will see details in this chapter that were not covered in Chapter 3.

## Host-Outbound Path: Packets Generated by the Routing Engine

Here, the same ping commands are executed from R1 and R3, but this time we focus on the outbound ICMP echo replies instead. As you can see in Figure 4.1, the R2's routing engine answers the previous 10pps of echo requests coming from routers R1 and R3.

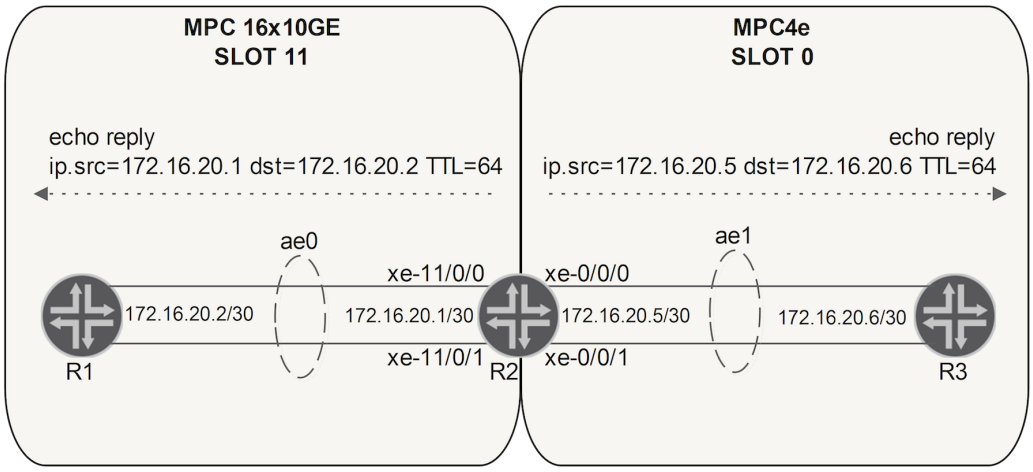



Figure 4.1 Outbound ICMP packets from R2

First of all, you have to realize that when the RE generates packets (like the ICMP echo replies in this example), it performs many tasks, including:

- Building the Layer 2 header of the packet. The RE builds itself the Layer 2 Ethernet header of the host outbound packets. This includes VLAN encapsulation.
- Outbound queue assignment. The RE sets two types of queue infomation in the packet. First, the TTP queue for the trip from RE to MPC. Second, the forwarding class, which later determines the relative (0-7) queue where the packet will be put before being sent out of the PFE.
- Forwarding next hop assignment. This means the IFD index on which the packet should be forwarded.

In the case of ECMP, the Routing Engine sends packets in a round-robin manner over the equal cost paths. In the LAG interface scenario, the Routing Engine will choose the lowest IFD index between child links. If you take the case of AE0 and AE1, which have two child links: xe-11/0/0 and xe-11/0/1, and xe-0/0/0 and xe-0/0/1, respectively, you can easily deduce on which interface the ICMP echo replies will be sent using these commands:

```
user@R2> show interfaces xe-11/0/[0-1] | match Index
  Interface index: 590, SNMP ifIndex: 773
  Interface index: 591, SNMP ifIndex: 727

user@R2> show interfaces xe-0/0/[0-1] | match Index
  Interface index: 571, SNMP ifIndex: 526
  Interface index: 572, SNMP ifIndex: 1079
```

Here, the lowest IFD index for AE0 member links is 590, which is the xe-11/0/0 interface. Similarly, the lowest IFD index for AE1 is found to be xe-0/0/0. This means that the RE notifies the µKernel of MPC 11 to send the ICMP reply to R1 via xe-11/0/0, and notifies MPC 0 to send the ICMP reply to R3 via xe-0/0/0. This information is also included in the TTP header.

## Host-Outbound Class of Service

Table 4.1 lists the default host-outbound queue assigned by RE/µKernel to common protocols. This default assignment may be altered by configuration, as explained later in this chapter.

NOTE The full table is kept up to date and is publicly available on the Juniper TechLibrary (go to www.juniper.net/documentation and search for *Default Queue Assignments for Routing Engine Sourced Traffic* to get the most current table).

Table 4.1 Default Outbound Queue Assigned for Each Protocol

| Host Protocol | Default Queue Assignment |
|---|---|
| Address Resolution Protocol (ARP) | Queue 0 |
| Bidirectional Forwarding Detection (BFD) Protocol | Queue 3 |
| BGP | Queue 0 |
| BGP TCP Retransmission | Queue 3 |
| Cisco HDLC and PPP | Queue 3 |
| Ethernet Operation, Administration, and Maintenance (OAM) | Queue 3 |
| SSH/Telnet/FTP | Queue 0 |
| IS-IS | Queue 3 |
| Internet Control Message Protocol (ICMP) | Queue 0 |
| LDP User Datagram Protocol (UDP) hello | Queue 3 |
| LDP keepalive and Session data | Queue 0 |
| LDP TCP Retransmission | Queue 3 |
| Link Aggregation Control Protocol (LACP) | Queue 3 |
| NETCONF | Queue 0 |
| NetFlow | Queue 0 |
| OSPF | Queue 3 |
| RSVP | Queue 3 |
| Routing Information Protocol (RIP) | Queue 3 |
| SNMP | Queue 0 |
| Virtual Router Redundancy Protocol (VRRP) | Queue 3 |

As mentioned above, the default queue assignment may be overridden by configuration:

```
set class-of-service host-outbound-traffic forwarding-class FC3
set class-of-service host-outbound-traffic dscp-code-point ef
```

NOTE This is the only method by which to modify queue assignment for non-IP and non-MPLS protocols like ISIS. Moreover, this method does not work for protocols managed by the MPC (delegated protocols).

These CLI knobs map the host-outbound traffic for all protocols to a given relative queue (actually forwarding class) and define the DSCP value to re-mark these host packets generated by the RE.

This job is all done by the RE. Indeed, the RE generates its packets with the right DSCP value (here EF) and then notifies, via an additional TLV inserted in the TTP header, which relative (0-7) HW queue of the MQ/XM WAN stream the packets should be put on.

Sometimes you want to classify some host protocols into a high priority queue and others to a best effort queue. To complete this selective per-protocol queue assignment you need to configure a specific firewall filter and apply it on the loopback interface in the output direction.

Here is a sample firewall filter configuration:

```
interfaces {
    lo0 {
        unit 0 {
            family inet {
                filter {
                    output HOST-OUTBOUND-REMARK;
}   }   }   }   }
firewall {
    family inet {
        filter HOST-OUTBOUND-REMARK {
            term ACPT-BGP {
                from {
                    protocol tcp;
                    port bgp;
                }
                then {
                    forwarding-class FC3;
                    dscp cs6;
                }
            }
            term ACPT-TELNET-SSH {
                from {
                    protocol tcp;
                    port [ telnet ssh ];
                }
                then {
                    forwarding-class FC1;
                    dscp af42;
                }
            }
            term ACCEPT-ALL {
                then accept;
}   }   }   }
```

In this example, only BGP and SSH/telnet protocols are reassigned to specific queues. This reclassification is also performed by the RE kernel. For the other protocols that only match the last "term," the RE uses the default queue assignment rules or the queue assigned by the global host-outbound-traffic knob.

NOTE In our case let's keep the default queue assignment (so the ICMP reply will be queued in the queue 0).

## From the RE Down to the MPC

The RE has built a L2 packet by adding a L2 header to the ICMP echo reply. The RE kernel already did all the work and it is not necessary to perform route lookup on the egress PFE. However, it still goes through the LU chip, in particular, to update counters and assign the absolute WAN output queue (based on the relative one previously set by the RE). For this reason, the RE flags the packet to bypass the route lookup on its way out.

Okay, once the packet is ready to send, the RE kernel sends it to the right MPC ( the MPC that hosts the output interface) via the RE's em0 interface as shown in Figure 4.2.

CAUTION The actual interface depends on the MX model, and even on the RE model. Check Chapter 3 of *This Week: A Packet Walthrough of M, MX and T Series* to see the method to determine the right interface to look at: http://www.juniper.net/us/en/training/jnbooks/day-one/networking-technologies-series/a-packet-walkthrough/ .

The Ethernet frame is encapsulated in a TTP header and then sent unicast to the µKernel of the egress MPC. The TTP header contains an additional TLV with WAN COS information (forwarding class).

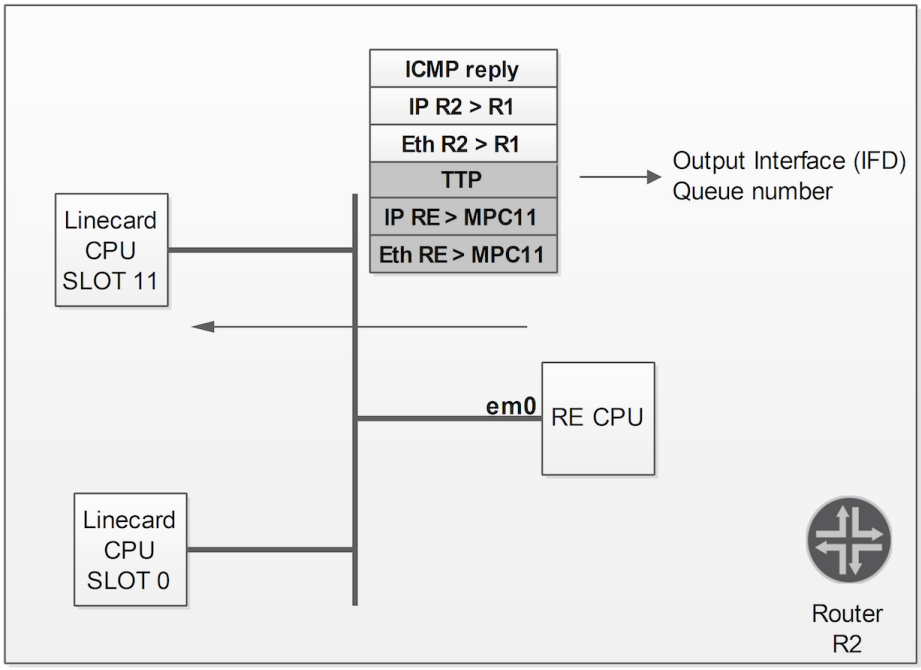



Figure 4.2 Host Outbound Packet Sent Over TTP

## From the MPC to the Outer World

The MPC microkernel decapsulates the original L2 Ethernet frame from TTP. The result is the original ICMP echo reply with its Ethernet header. The COS information that was present in the removed TTP header is then added (in a disposable header) to the L2 frame before sending it to the PFE.

Packets are then received in the µKernel's software queues before sending them to the PFE. The next command shows you that the echo reply packets received from RE (Layer 2 packets) are all in queue 0:

```
NPC11(R2 vty)# show ttp statistics
TTP Statistics:
[...]

TTP Transmit Statistics:
                   Queue 0     Queue 1     Queue 2    Queue 3
                 ----------  ----------  ----------  ----------
 L2 Packets        6100386           0           0           0
 L3 Packets              0           0           0           0

[...]
```

You can view host outbound packets caught by a specific MPC with this PFE command (from µKernel's point of view):

```
NPC11(R2 vty)# show pfe host-outbound packets

ifl input: 0
ifl output: 131075
ifd output: 590
Proto: 0 Hint:0x1 Qid:0 Cos:0
  00 00 01 75 80 00 00 21 - 59 a2 ef c2 00 21 59 a2    ...u...!Y....!Y.
  ef c0 08 00 45 00 00 54 - 6d 90 00 00 40 01 4b 07    ....E..Tm...@.K.
  ac 10 14 01 02 00 00 01 - 08 00 68 8e a3 d0 01 47    ..........h....G
  53 8d 95 de 00 03 6f 6f - 44 41 59 4f 4e 45 20 44    S.....ooDAYONE D
  00 00 00 00 00 01 00 00 - 00 00 00 00 00 00 00 02    ................
  00 03 02 4e 00 00 01 75 - 80 00 00 21 59 a2 ef c2    ...N...u...!Y...
  00 21 59 a2 ef c0 08 00 - 45 00 00 54 6d 9a 00 00    .!Y.....E..Tm...
  40 01 4a fd ac 10 14 01 - 02 00 00 01 08 00 de 18    @.J.............
   @.J.............
```

The µKernel of the MPC passes the packet to the MQ/XM chip (without TOE). As mentioned previously, the host path is viewed as a WAN interface and therefore from the perspective of MQ/XM the packet coming from this pseudo-WAN interface is handled by the WAN Input functional block (as many other physical interfaces).

Also remember, the µKernel has PCI interfaces to the MQ/XM chip. In this case, the µKernel sends the packet to the WI functional block via PCIe DMA. WI puts host packets in a specific WAN stream: actually the host stream (ID 1151) already seen in the host inbound path (stream 1151 is bidirectional).

But how does the µKernel choose the right MQ/XM chip?

Indeed, a single MPC hosts several PFEs. The RE sends some useful information via the TTP header to the MPC and some of it is the Output Interface (the output IFD). The µKernel performs a kind of IFD lookup to find to which ASICs the packet has to be sent. You can use the PFE command to retrieve this information for the xe-11/0/0 and xe-0/0/0 interfaces:

```
user@R2> request pfe execute target fpc11 command "show ifd 590" | match PFE | trim 5
PFE:  Local, PIC:  Local, FPC Slot: 11, Global PFE: 44, Chassis ID: 0
```

```
user@R2> request pfe execute target fpc0 command "show ifd 571" | match PFE | trim 5
  PFE:  Local, PIC:  Local, FPC Slot: 0, Global PFE: 0, Chassis ID: 0
```

As previously mentioned, the WI functional block manages these host packets as WAN input packets. Because it knows the packet is coming from the host it puts it on Host Stream 1151. One counter of the WI block is by default programmed to count packets in Host Stream 1151. Let's check on the WI statistics of the Host Stream. On the MQ chip:

```
NPC11(R2 vty)# show mqchip 0 wi stats
WI Counters:
                  Counter       Packets     Pkt Rate          Bytes    Byte Rate
  ---------------------- ------------- ------------ -------------- ------------
    RX Stream 1025 (001)             0            0              0            0
    RX Stream 1026 (002)        630787            9       62935934          950
    RX Stream 1027 (003)             0            0              0            0
    RX Stream 1151 (127)      10161965           10      817609234         1117

    DROP Port 0 TClass 0             0            0              0            0
    DROP Port 0 TClass 1             0            0              0            0
    DROP Port 0 TClass 2             0            0              0            0
    DROP Port 0 TClass 3             0            0              0            0
```

And on the XM chip:

```
NPC0(R2 vty)# show xmchip 0 wi stats 0 <<< second 0 means WAN Input Block 0

WI statistics (WAN Block 0)
---------------------------

Tracked stream statistics
-------------------------
Track Stream Stream Total Packets     Packets Rate    Total Bytes        Bytes Rate
      Mask   Match                    (pps)                              (bps)
-------------------------------------------------------------------------------------
0     0x7c   0x0    4688             10              449268             8000
1     0x7c   0x4    2147483648       0               137438953472       0
2     0x7c   0x8    5341224839       0               454111053838       0
3     0x7c   0xc    2147483648       0               137438953472       0
4     0x0    0x0    3212591566       23              317782816860       17936
6     0x0    0x0    3212591566       23              317782816860       17936
7     0x0    0x0    3212591566       23              317782816860       17936
8     0x0    0x0    3212591566       23              317782816860       17936
9     0x0    0x0    3212591566       23              317782816860       17936
10    0x7f   0x7f   10053356         10              798303776          9392
11    0x70   0x60   0                0               0                  0
-------------------------------------------------------------------------------
```

The host-outbound L2 packets generated by the RE (in our case, the ICMP echo reply packet with its L2 header) carry a flag to bypass the route lookup phase in LU chip.

As you've seen in the last section of Chapter 2 (covering egress PFE forwarding), the WI takes a chunk of this packet and gives the parcel to the LO functional block which finally passes the parcel to the LU chip. The parcel has the "bypass lookup" flag so the LU chip just does a very simple task. LU selects the right physical WAN Output Stream based on the output IFD (or IFL) and the forwarding class previously set by the RE. Then the LU returns the parcel to the LI functional block of MQ or XM, embedding the WAN Output Queue (The Absolute Queue number) in the L2M header.

On the MQ Chip:

```
NPC11(R2 vty)# show mqchip 0 ifd
[…]
  Output   IFD     IFD             Base
  Stream  Index    Name     Qsys   Qnum
  ------ ------ ---------- ------ ------
    1024    590  xe-11/0/0    MQ0      0 <<<< The IFD 590 is assigned to WO STREAM 1024
    1025    591  xe-11/0/1    MQ0      8
    1026    592  xe-11/0/2    MQ0     16
    1027    593  xe-11/0/3    MQ0     24
```

And on the XM Chip:

```
NPC0(R2 vty)# show xmchip 0 ifd list 1 <<<< 1 means Egress direction

Egress IFD list
---------------

------------------------------------------------------------------
IFD name     IFD    PHY     Scheduler  L1 Node  Base   Number of
             Index  Stream                      Queue  Queues
------------------------------------------------------------------
xe-0/0/0     571    1024    WAN        0        0      8  <<<< The IFD 571 = WO STREAM 1024 (the
                                                           same value as on MQ: pure coincidence)
xe-0/0/1     572    1025    WAN        1        8      8
xe-0/0/2     573    1026    WAN        2        16     8
xe-0/0/3     574    1027    WAN        3        24     8
et-0/1/0     575    1180    WAN        4        32     8
------------------------------------------------------------------
```

MQ/XM handles the parcel via the LI block, then strips the L2M header, stores the parcel, and then gives the packet's pointer to the SCHED block. The scheduler associated to the IFD does its job. You can retrieve L1/L2/Q node information by using one of the already used commands (see Appendix A for more information).

Here's an example of retrieving that information for interface xe-11/0/0 (IFD 590):

```
NPC11(R2 vty)# show cos halp ifd 590 1 <<< 1 means egress direction

------------------------------------------------------------------------------
IFD name: xe-11/0/0   (Index 590)
    MQ Chip id: 0
    MQ Chip Scheduler: 0
    MQ Chip L1 index: 0
    MQ Chip dummy L2 index: 0
    MQ Chip base Q index: 0
    Number of queues: 8
    Rich queuing support: 0 (ifl queued:0)
Queue    State        Max      Guaranteed   Burst Weight Priorities Drop-Rules
Index                 rate        rate      size           G    E   Wred  Tail
------ ----------- ---------- ---------- ------ ------ ---------- ----------
     0  Configured   222000000   277500000  32767     62   GL   EL     4   145
     1  Configured   555000000   111000000  32767     25   GL   EL     4   124
     2  Configured   555000000   111000000  32767     25   GL   EL     4   124
     3  Configured   555000000    Disabled  32767     12   GH   EH     4    78
     4  Configured   555000000           0  32767      1   GL   EL     0   255
     5  Configured   555000000           0  32767      1   GL   EL     0   255
     6  Configured   555000000           0  32767      1   GL   EL     0   255
     7  Configured   555000000           0  32767      1   GL   EL     0   255
```

Remember that our ping echo reply is sent to Queue 0 (by default). You can even use the following PFE commands to retrieve Q node information respectively for the MQ and XM chips:

```
show mqchip 0 dstat stats 0 0
show xmchip 0 q-node stats 0 0
```

Let's check the interface queue statistics, using the CLI:

```
user@R2>
Physical interface: xe-11/0/0, Enabled, Physical link is Up
  Interface index: 590, SNMP ifIndex: 773
  Description: LAB_DAV_R2
Forwarding classes: 16 supported, 4 in use
Egress queues: 8 supported, 4 in use
Queue: 0, Forwarding classes: FC0
  Queued:
    Packets              :              5650682                   10 pps
    Bytes                :            678761824                 9664 bps
  Transmitted:
    Packets              :              5650682                   10 pps
    Bytes                :            678761824                 9664 bps
    Tail-dropped packets :                    0                    0 pps
```

All looks well. Great! Before leaving the router, the WO functional block reassembles the packets, sends it to the MAC controller, and the packet finally leaves the router. As you have seen previously you can use PFE commands to check WO statistics for a given WAN stream and also check the MAC controller output statistics. Hereafter, just a quick reminder of these PFE commands without their output. We take as example the interfaces xe-11/0/0 (WAN Stream 1024) and xe-0/0/0 (WAN Stream 1024).

For xe-11/0/0:

```
test mqchip 0 counter wo 0 1024 <<< 0 1024 means activate counter 0 for Stream 1024
show mqchip 0 counters output stream 1024
test mqchip 0 counter wo 0 default
show mtip-xge summary
show mtip-xge 5 statistics <<< 5 is the index of xe-11/0/0 retrieved by the previous command
```

And for xe-0/0/0:

```
test xmchip 0 wo stats stream 0 1024 <<< 0 1024 means activate counter 0 for Stream 1024
show xmchip 0 phy-stream stats 1024 1 <<< 1 means Output direction
test xmchip 0 wo stats default 0 0
show mtip-cge summary
show mtip-cge 5 statistics <<< 5 is the index of xe-0/0/0 retrieved by the previous command
```

Figure 4.3 illustrates this PFE host outbound path.

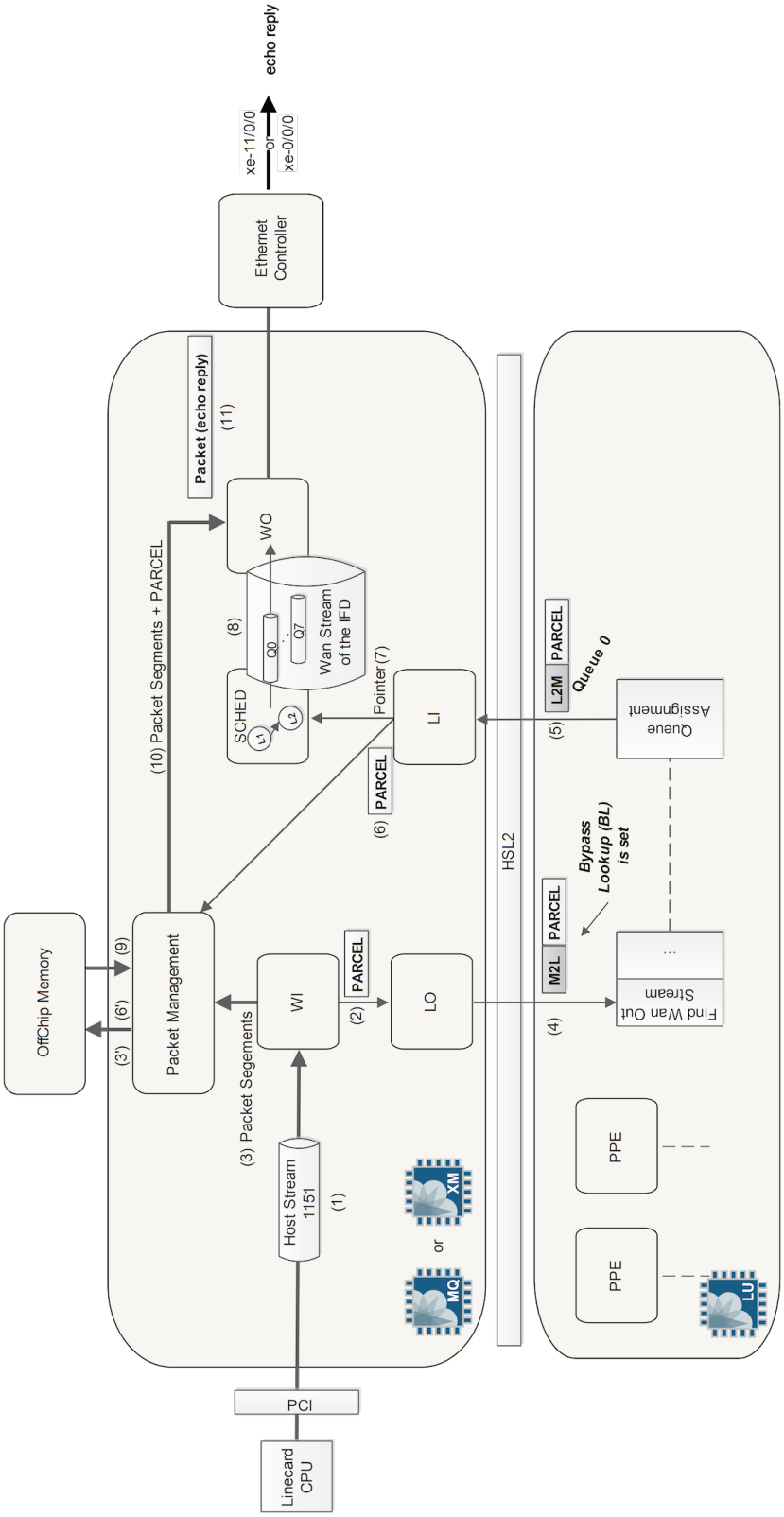



Figure 4.3 Host Outbound Path at the PFE Level

While this first host packet walkthrough analysis was long almost all the functional blocks were covered and that same analysis procedure can easily be done for any other Layer 3 or Layer 2 host packets such as BGP, LDP, ISIS, and so on.

Now let's see a specific scenario where the packets are not generated by the Routing Engine, but by the Line Card itself.

## Host-Outbound Path: Packets Generated by the MPC Line Card

This last section extends our topic of Host traffic to focus on a short case study on distributed control plane protocols. What does that mean?

For some protocols, the Routing Engine delegates their management to the MPC's µKernel CPU. Therefore, the protocol's state machine is managed by the µKernel, which only sends feedback to the RE when the state changes and for certain protocol's statistics (via Inter Process Communication, or IPC, which is transported as TCP/IP in the internal Ethernet network).

NOTE There are other cases where the MPC's µKernel CPU generate their own host-outbound packets, for example, ICMP Unreachables sent as a response to incoming exceptional (like TTL=1) transit traffic. This stateless example is not covered here, but the internals are the same.

For this case study, let's activate LACP (fast timer and active mode) on AE1 between the R2 and R3 routers and then follow how LACP is managed. Other protocols like BFD, Ethernet OAM or VRRPv3 can also be delegated in this manner.

Incoming LACP goes through a similar path as ICMP or ARP packets. It's an exception packet for the LU chip that is identified as a Control packet (PUNT 34 code):

```
NPC0(R2 vty)# show jnh 0 exceptions terse
[...]
Routing
----------------------
control pkt punt via nh            PUNT(34)            24        2640
```

However, LACP's protocol data units (PDUs) are not sent to the Routing Engine but they are fully processed by the µKernel itself. To confirm this point, let's try to monitor non-IP traffic on AE1. You should see nothing:

```
user@R2> monitor traffic interface ae1 no-resolve matching "not ip"
verbose output suppressed, use <detail> or <extensive> for full protocol decode
Address resolution is OFF.
Listening on ae1, capture size 96 bytes

0 packets received by filter
```

The protocol delegation depends on the version of Junos and the type of hardware you are using, but to know which protocol is distributed on a linecard you can execute the following command, this time on MPC in slot 0 (remember the Junos release is 14.1):

```
NPC0(R2 vty)# show ppm info
Status: Connected to ppmd
Protocol: OSPF2  Support: All Status: Ready
Refresh time: 15000 do_not_refresh = 0
Protocol: OSPF3  Support: All Status: Not-ready
Refresh time: 15000 do_not_refresh = 0
```

```
Protocol: ISIS  Support: All Status: Ready
Refresh time: 15000 do_not_refresh = 0
Protocol: BFD  Support: All Status: Ready
Refresh time: 15000 do_not_refresh = 8
Protocol: LDP  Support: All Status: Ready
Refresh time: 15000 do_not_refresh = 0
Protocol: STP  Support: All Status: Ready
Refresh time: 15000 do_not_refresh = 0
Protocol: LFM  Support: All Status: Ready
Refresh time: 15000 do_not_refresh = 0
Protocol: CFM  Support: All Status: Ready
Refresh time: 15000 do_not_refresh = 0
Protocol: LACP  Support: All Status: Ready
Refresh time: 15000 do_not_refresh = 0
Protocol: VRRP  Support: All Status: Not-ready
Refresh time: 15000 do_not_refresh = 0
Protocol: RPM  Support: All Status: Ready
Refresh time: 15000 do_not_refresh = 0
```

Now, let's check DDOS LACP's policer statistics on MPC 0:

```
NPC0(R2 vty)# show ddos policer lacp stats
DDOS Policer Statistics:

                                                              arrival   pass  # of
idx prot    group        proto on     loc    pass     drop    rate   rate flows
---  ---  -------  ---------- --  ------  ------   ------  ------ ------ -----
111 2c00     lacp    aggregate  Y   UKERN     174        0       2      2     0
                                    PFE-0     174        0       2      2     0
                                    PFE-1       0        0       0      0     0
```

Why 2pps of LACP while LACP with fast time sends one LACP PDU per second?

Remember that AE1 has two child links. This is why PFE 0 of MPC 0, which connects the two child links of AE1, sees 2pps and delivers these two packets to the µKernel every second.

Before moving on to the analysis of the distributed protocol management, let's quickly practice by retrieving the two LACP packets per second queued at the XM chip level (AE1 has child links connected to MPC4e card).

First of all, you need to find the queue assigned to LACP packets by the LU chip:

```
NPC0(R2 vty)# show ddos asic punt-proto-maps
[...]
         ------------------------------------------------------------------
         type   subtype            group proto          idx q# bwidth  burst
         ------ ----------    ---------- ----------   ---- -- ------ ------
         contrl LACP                lacp aggregate    2c00  3  20000  20000
[...]
```

You can see that Queue 3 is used for LACP. Now retrieve the base Queue Node ID of the Host stream (remember = 1151):

```
NPC0(R2 vty)# show cos halp queue-resource-map 0 <<<< 0 means PFE_ID = 0
[...]
  ------------------------------------------------------------

  stream-id  L1-node  L2-node  base-Q-Node
  ------------------------------------------------------------
       1151      126      252         1008  <<< Base queue
[...]
```

Finally, collect the statistics of q-node 1011 (base queue plus 3), the queue that the LU chip assigns LACP packets to:

```
NPC0(R2 vty)# show xmchip 0 q-node stats 0 1011  <<<<2nd 0 is QSys 0 and 1011 = 1008+3

Queue statistics (Queue 1011)
-----------------------------

------------------------------------------------------------------------
Color  Outcome              Counter  Counter  Total                Rate
                            Index    Name
------------------------------------------------------------------------
All    Forwarded (No rule)  228      Packets  0                    0 pps
All    Forwarded (No rule)  228      Bytes    0                    0 bps
All    Forwarded (Rule)     229      Packets  5623                 2 pps  <<< The 2pps LACP
All    Forwarded (Rule)     229      Bytes    838236               2384 bps
All    Force drops          232      Packets  0                    0 pps
All    Force drops          232      Bytes    0                    0 bps
All    Error drops          233      Packets  0                    0 pps
All    Error drops          233      Bytes    0                    0 bps

[...]
```

And you can see the 2pps. Okay, let's have a look at who manages the distributed protocols.

The PPMd (Point to Point Management Daemon) process on the Routing Engine is in charge of managing the periodic "messages" of many protocols (such as Keepalives, Hello PDUs, etc.). It sends back to the routing daemon (RPD) information when there is a state change of a protocol.

PPMd in the RE peers with a µKernel thread in the line card, called ppman, where man stands for manager. This thread handles the periodic messages of distributed protocols, and notifies the Routing Engine when there is a change in the state of an adjacency.

```
NPC0(R2  vty)# show  thread
PID PR State     Name                   Stack Use  Time (Last/Max/Total) cpu
--- -- -------   --------------------   ---------  --------------------
[...]
 53 M  asleep    PPM Manager           4256/8200   0/0/19 ms  0%
 54 M  asleep    PPM Data thread       1584/8200   0/0/375 ms  0%
[...]
```

For a given MPC, and a given distributed protocol, you can check how many adjacencies are handled by the µKernel in real time. Let's try an example on MPC 0:

```
NPC0(R2 vty)# show ppm adjacencies protocol lacp

PPM Adjacency Information for LACP Protocol

IFL Index  Holdtime     PPM handle
            (msec)
360        3000         11
363        3000         15

Total adjacencies: 2
```

The above output shows you that MPC 0 has two LACP adjacencies (AE1 is made of two links). The IFL column helps you to retrieve the mapping between the PPM handle and the interface:

```
user@R2> show interfaces xe-0/0/*.0 | match "index"
  Logical interface xe-0/0/0.0 (Index 360) (SNMP ifIndex 1022)
  Logical interface xe-0/0/1.0 (Index 363) (SNMP ifIndex 1123)
```

Finally, you can call another command to collect statistics of a given distributed protocol:

```
NPC0(R2 vty)# show ppm statistics protocol lacp

LACP Transmit Statistics
========================
Packets Transmitted   : 186
Packets Failed to Xmit: 0

LACP Receive Statistics
=======================
Total Packets Received      : 150
Packets enqued to rx Q      : 150
Packets dequeued by lacp    : 150
Packets Send to RE          : 10 <- Notif. Sent to RE (8 + 2)
  No (pfe) interface found  : 0
  Conn not ready            : 0
  No (conn) interface found : 0
  No adjacency found        : 2  <- Notif. Sent to RE
  Packet content change     : 8  <- Notif. Sent to RE
Packets Absorbed            : 140
Packets Dropped(Invalid)    : 0

Interface name  xe-0/0/0.0,  Pkt tx: 54, Tx errors: 0, Pkt rx: 54,
 Pkt absorbed: 54, No Adj: 0, Pkt change: 0

Interface name  xe-0/0/1.0,  Pkt tx: 57, Tx errors: 0, Pkt rx: 21,
 Pkt absorbed: 16, No Adj: 1, Pkt change: 4
```

You can see that distributing protocols in a MPC can reduce a lot of control plane traffic toward the RE, allowing you to scale more in terms of LACP, or BFD, OAM CFM/LFM, VRRPv3, etc. adjacencies.

This ends Chapter 4's packet walkthrough that involved the host outbound path. The next chapter focuses on the multicast traffic.

# Chapter 5

## Replication in Action

This last chapter of *This Week: An Expert Packet Walkthrough on the MX Series 3D* focuses on a complex topic: multicast. It only covers the case of IPv4 SSM (Single Source Multicast) replication in an effort for you to understand how the multicast replication tree is built and updated in the MX Series 3D.

Let's assume that R2 is configured with the `enhanced-ip` feature set at the `chassis/network-services` level. This mode allows us to use all the optimizations and robustness features that apply to multicast forwarding, at what has become the familiar PFE level.

NOTE Fabric-less routers like MX5, MX10, MX80, or MX104, and also MX2010 and MX2020, have enhanced-IP active regardless of the knob configuration.

## Multicast Network Topology

First, our base topology includes some changes to account for the multi-PFE replication scenario. The R4 router has moved to another interface of R2 (now xe-11/3/0), and the topology has added a directly-attached receiver onto the xe-11/1/0 interface of R2. Figure 5.1 shows you the topology.

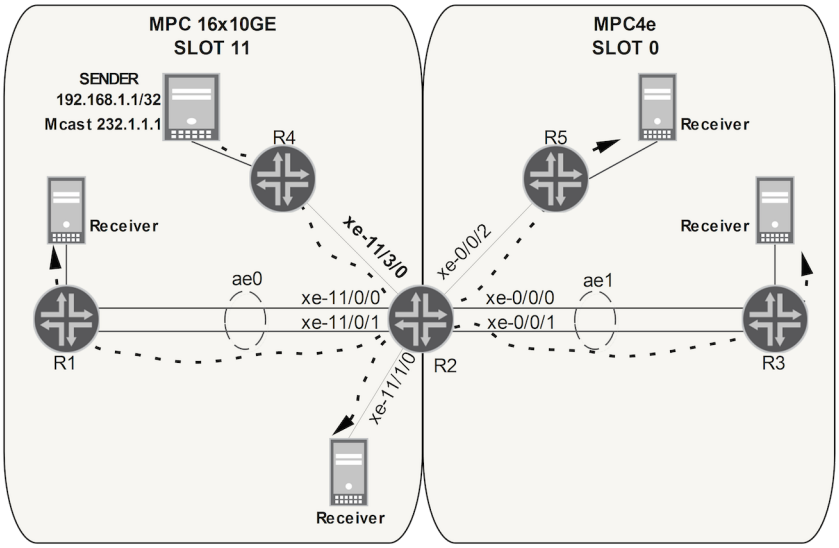



Figure 5.1 Multicast Topology

You can see in Figure 5.1 that R2 receives the multicast stream (232.1.1.1/192.168.1.1) on its xe-11/3/0 interface and replicates the stream four times in order to reach the four receivers.

## IP Multicast Control Plane

Protocol Independent Multicast (PIM) sparse mode is the protocol that makes it possible to build the multicast distribution tree in IPv4 or in IPv6. The distribution tree spans from the source of a stream to a set of multicast subscribers or receivers.

A multicast stream is uniquely identified by a couple of addresses named (S,G) - (source,group):

- The source IP address: is a unicast IPv4 or IPv6 address.

- The destination IP address is a multicast group: an IP4 address in the range 224/4, or an IPv6 address in the range FF00::/8.

Usually, final subscribers don't know the source(s) of a given multicast stream, they only know the group address. They use a protocol like IGMPv2 or MLDv1 that simply provides messages to *join* or *leave* a given multicast group G.

NOTE Enhanced protocols like IGMPv3 or MLDv2 provide the ability to also convey the associated source S of a given group G (but is not within the scope of this book).

Therefore, IGMPv2 or MLDv1 protocols convey ASM information (Any Source Multicast). Its notation is (*,G) and refers to any source that sends traffic to the multicast G address. In contrast, SSM information (Source Specific Multicast) refers to a specific stream (S,G).

The last-hop PIM router receives group information from subscribers and translates it into upstream ASM or SSM PIM messages:

- IGMP Report / MLD Listener Report messages are translated into PIM Join/Prune messages with details in its join list. These are typically called PIM Join messages.
- IGMP Leave / MLD Listener Done messages are translated into a PIM Join/Prune message with details in its prune list. These are typically called PIM Prune messages.

Our topology focuses on SSM. Consider that border routers directly attached to multicast receivers have a static mapping between the multicast group 232.1.1.1 and the multicast source 192.168.1.1/32 (this is a SSM-map entry). Thus a router which receives an IGMPv2 report for 232.1.1.1 can directly send a PIM SSM Join for (232.1.1.1; 192.168.1.1) along the Source Path Tree (SPT).

## IP Multicast Forwarding Plane

When a multicast stream is received by a router, the device first checks that the packets are being received on the interface used to reach the source. This mechanism is called a RPF check (Reverse Path Forwarding). If the multicast packet fails the RPF check it is silently dropped (and interface mismatch statistics are increased). If it passes the check point, the router replicates and forwards the packet copies. Hop-by-hop the multicast stream follows the multicast tree replication to reach all its receivers.

The previous network image can be ported to the MX router replication model. Let's view the MX as a network where the nodes are the PFEs and the network links are actually the fabric links.

Now, we can identify two types of replication within a chassis:

- Intra-PFE replication: a PFE replicates the packet to a set of multicast OIFs (outgoing interfaces) that are local to the PFE itself.
- Inter-PFE replication: a PFE needs to send one copy of the packet to one or more remote PFEs that contain multicast OIFs for the packet.

In the case of Intra-PFE replication the LU chip generates a Parcel per local OIF, in order for the MQ to craft the outgoing packet by appending each OIF Parcel and the remaining buffered data – should the packet be greater than 320 bytes – (see figure 5.5, PFE 0).

Inter-PFE replication is slightly more complex: in this case the ingress PFE computes the dynamic tree, based on information received from the RE. When the ingress PFE forwards packets to the remote PFEs, the internal tree is embedded with each packet (in the FAB header). This allows on-the-fly tree updates and no need to syncrhonize PFEs.

Note that between one given (source PFE, destination PFE) pair, only one copy of each packet is sent: further replication is handled by the destination PFE (to local OIFs and/or to subsequent PFEs). Said that, there are two methods to internally replicate multicast among PFEs. The replication method used depends on the MPC model and also on the configuration of the chassis network-services mode (let's assume enhanced-IP). These two methods can co-exist.

The two ways to replicate multicast are:

- Binary Tree Replication (the only mode supported by MPC1 and MPC2): Each PFE can replicate packets towards one or two other PFEs.
- Unary Tree Replication: Each PFE can only forward multicast to another PFE. This method saves fabric link bandwidth and is the one used in high-capacity MPCs (from MPC 16x10GE onwards).

NOTE Without enhanced-IP mode all boards do binary replication. On-the-fly is supported for both replication methods.

Figure 5.2 illustrates these two methodologies using the topology of Figure 5.1 as the base.

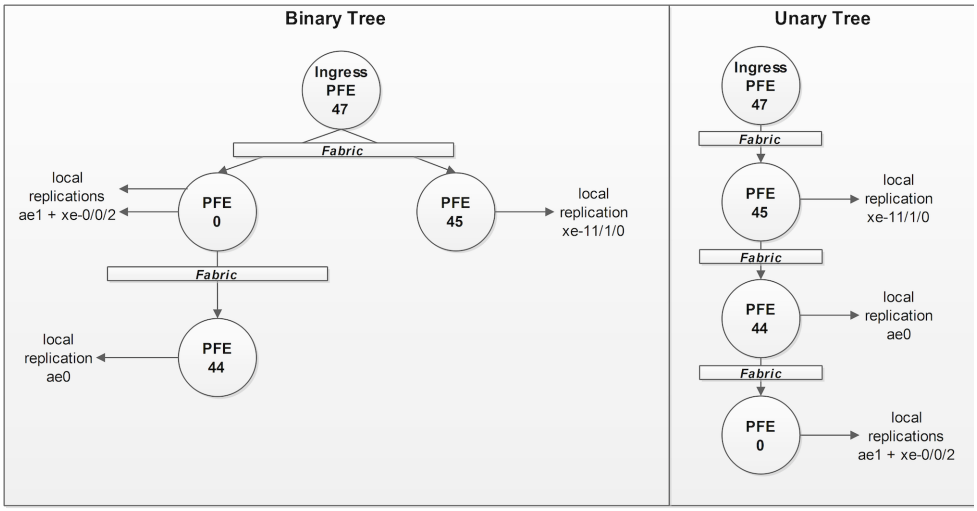



Figure 5.2 Multicast Replication Methods

As you can see binary tree replication consumes more fabric bandwidth (between PFE and the Fabric) than the unary tree replication, which saves more bandwidth and scales more.

MPC1 and MPC2 support only binary tree replication, but the other MPCs support both methods. Nevertheless, unary tree replication requires the enhanced-ip knob to be explicitly configured:

```
chassis {
    network-services enhanced-ip;
}
```

For the following section on building a multicast entry, let's assume that R2 is configured with this knob and only the case of unary tree replication will be covered.

## Building a Multicast Forwarding Entry

Let's start by checking the multicast entry creation at the RIB and FIB level on R2 router and let's assume that the multicast source has started a 20Kpps stream and all the receivers have joined the stream.

Now, if you check the PIM Join state of R2, you should see that the stream (232.1.1.1;192.168.1.1) is replicated four times:

```
user@R2> show pim join extensive
[...]
Group: 232.1.1.1
    Source: 192.168.1.1
    Flags: sparse,spt
    Upstream interface: xe-11/3/0.0  <<<< Incoming Interface (RPF Check)
    Upstream neighbor: 172.16.20.10
    Upstream state: Join to Source
    Keepalive timeout: 0
    Uptime: 00:07:53
    Downstream neighbors:
        Interface: ae0.0   <<<<<<<< 1 replication - to R1
            172.16.20.2 State: Join Flags: S Timeout: 156
            Uptime: 00:07:53 Time since last Join: 00:00:53
        Interface: ae1.0   <<<<<<<< 1 replication - to R2
            172.16.20.6 State: Join Flags: S Timeout: 205
            Uptime: 00:01:04 Time since last Join: 00:00:04
        Interface: xe-0/0/2.0 <<<<<< 1 replication - to R5
            172.16.20.14 State: Join Flags: S   Timeout: 164
            Uptime: 00:00:12 Time since last Join: 00:00:12
        Interface: xe-11/1/0.0 <<<<<< Replication to local receiver
            172.16.20.254 State: Join Flags: S   Timeout: Infinity
            Uptime: 00:40:56 Time since last Join: 00:40:56
    Number of downstream interfaces: 4
```

That looks good. Now let's check the forwarding cache entry:

```
user@R2> show multicast route extensive
Group: 232.1.1.1
    Source: 192.168.1.1/32
    Upstream interface: xe-11/3/0.0  <<< RPF Check
    Downstream interface list:
       ae0.0 ae1.0 xe-0/0/2.0 xe-11/1/0.0   <<< Outgoing Interface List (OIL)
    Number of outgoing interfaces: 4
    Session description: Source specific multicast
    Statistics: 10380 kBps, 20000 pps, 6829154686 packets
    Next-hop ID: 1048586  <<< Indirect NH
    Upstream protocol: PIM
    Route state: Active
    Forwarding state: Forwarding
    Cache lifetime/timeout: forever <<< infinite for SSM route
    Wrong incoming interface notifications: 0 <<< number of RPFs that failed
    Uptime: 00:16:28
```

You can again see the RPF interface, the OIL (Outgoing Interface List). This entry is kept in the RIB and FIB (as long as PIM Joins are maintained): this is because the stream is an SSM one. For ASM entries, data plane (real traffic) must be present to keep a forwarding entry alive.

Now, let's have a look at the next hop ID. What does it mean?

Actually, this is an indirect-next hop which refers to the Outgoing Interface List (OIL) – the next hop hierarchy is covered in detail later. Each OIL has a dedicated indirect next hop (the entry point of the multicast replication tree).

You can check the next hop list with the following command and you'll find back the next hop referring to the OIL:

```
user@R2> show multicast next hops
Family: INET
ID          Refcount KRefcount Downstream interface
1048586            2          1 ae0.0
                              ae1.0
                              xe-0/0/2.0
                              xe-11/1/0.0
```

How to interpret these different entries? Actually, these different outputs present almost the same thing: information regarding an IPv4 multicast route. For Junos, an IPv4 multicast route is a /64 prefix made of the group address and the source address. IPv4 Multicast routes are stored on inet.1 routing table:

```
user@R2> show route table inet.1 detail
[...]
232.1.1.1, 192.168.1.1/64 (1 entry, 1 announced)
        *PIM    Preference: 105
                Next hop type: Multicast (IPv4) Composite, Next hop index: 1048586
```

Let's now check how this multicast route is "programmed" in the FIB. First execute the following CLI command:

```
user@R2> show route forwarding-table multicast destination 232.1.1.1 extensive
[...]
Destination:  232.1.1.1.192.168.1.1/64
  Route type: user
  Route reference: 0                   Route interface-index: 378
  Multicast RPF nh index: 0
  Flags: cached, check incoming interface , accounting, sent to PFE, rt nh decoupled
  Next-hop type: indirect              Index: 1048586  Reference: 2
  Nexthop:
  Next-hop type: composite             Index: 862      Reference: 1
  Next-hop type: unicast               Index: 1048579  Reference: 2
  Next-hop interface: ae0.0
  Next-hop type: unicast               Index: 1048580  Reference: 2
  Next-hop interface: ae1.0
  Next-hop type: unicast               Index: 1048583  Reference: 2
  Next-hop interface: xe-0/0/2.0
  Next-hop type: unicast               Index: 1048585  Reference: 2
  Next-hop interface: xe-11/1/0.0
```

A multicast route in the FIB is made of several things:

- The RPF interface information used for the RPF check. This is the route interface-index (IFL 378 which is interface xe-11/3/0.0):

```
user@R2> show interfaces xe-11/3/0.0 | match index
  Logical interface xe-11/3/0.0 (Index 378) (SNMP ifIndex 856)
```

- The next hop chain whose entry point is the indirect-next hop – 1048586 points to a composite next hop 862, which is a list of unicast next hops (OIL) : the four outgoing interfaces.

Let's move on with the PFE point of view (MPC in slot 11) and check back the multicast FIB entry:

```
NPC11(R2 vty)# show route long_ip prefix 232.1.1.1.192.168.1.1/64
Destination                       NH IP Addr      Type     NH ID Interface
--------------------------------- --------------- -------- ----- ---------
232.1.1.1.192.168.1.1/64                          Indirect 1048586 RT-ifl 378

Nexthop details:
1048586(Indirect, IPv4, ifl:0:-, pfe-id:0, i-ifl:0:-)
    862(Compst, IPv4, ifl:0:-, pfe-id:0, comp-fn:multicast)
        1048579(Aggreg., IPv4, ifl:341:ae0.0, pfe-id:0)
            755(Unicast, IPv4, ifl:367:xe-11/0/0.0, pfe-id:44)
            756(Unicast, IPv4, ifl:368:xe-11/0/1.0, pfe-id:44)
        1048580(Aggreg., IPv4, ifl:342:ae1.0, pfe-id:0)
            757(Unicast, IPv4, ifl:372:xe-0/0/0.0, pfe-id:0)
            758(Unicast, IPv4, ifl:373:xe-0/0/1.0, pfe-id:0)
        1048583(Unicast, IPv4, ifl:374:xe-0/0/2.0, pfe-id:0)
        1048585(Unicast, IPv4, ifl:380:xe-11/1/0.0, pfe-id:45)
```

As you can see the chain is a bit longer. Why?

Actually, LAG interfaces are not a real unicast next hop. They are an aggregate next hop made of a list of unicast next hops (one per child link) that could be spread over several PFEs (this not the case here).

But the multicast traffic is not replicated across all the child links: only one link is chosen, and there is load balancing. Forwarding over LAG outgoing interfaces is managed like unicast traffic. The same fields are taken into account to compute hash keys in order to load balance multicast over child links of a LAG interface.

## Multicast Replication

Now, let's move on with the multicast replication. Multicast lookup is performed by the ingress LU chip in several steps:

- Step 1: RPF Check – remember the multicast route in the FIB includes the RT IFL information.
- Step 2: Extract the OIL (this is the composite next hop associated with the multicast route G.S/64)
- Step 3: If some outgoing interfaces are a LAG, trigger hash key computation to select one link among the child links of the LAG interface. Repeat this step for each outgoing LAG interface in the outgoing interface list.
- Step 4: Build the distribution mask to indicate which PFE is interested to receive the data: in other words, the list of PFEs that should receive a copy of the multicast stream.

The PFE list is a PFE bit mask of 32 or 64 bits. Each bit represents the global PFE ID: Bit 0 = PFE_ID 0 ; bit 1 = PFE_ID 1, and so on. Let's use our example where OIL includes PFE_ID: 44, 45, 0, which are the global PFE IDs for (MPC 11, PFE 0), (MPC 11, PFE 1), and (MPC 0, PFE 0). In this case the PFE mask will be a word of 64 bits because there are PFE_IDs above 31. Figure 5.3 shows you the PFE Mask that should generate the ingress PFE 47 (attached to incoming interface xe-11/3/0):

REMEMBER

In Junos you assume, at most, four PFE per slots and MX960 is a 12-slot chassis. So the highest PFE ID is 47 (12x4 - 1). For MX2020, which has 20 slots, Junos adds an extended mask (longer than the default 32- or 64-bit mask) to accomodate the number of PFEs.

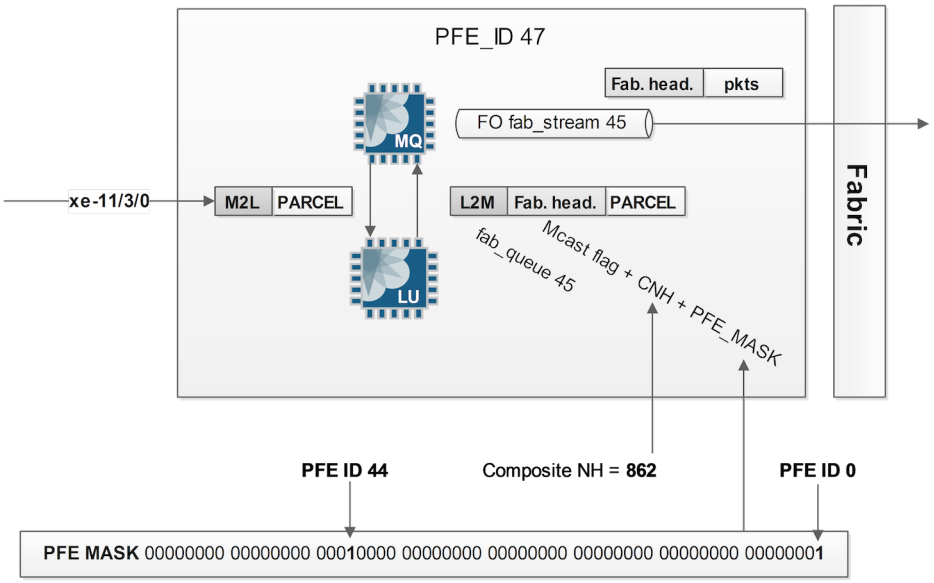



Figure 5.3 PFE Mask Computation

Something is strange, isn't it? Why is the bit 45 not set (PFE ID 45)? Actually PFE_ID 45 is the highest PFE ID in the PFE list, so this PFE will be the first PFE to receive the multicast copy from the ingress PFE. Remember, when a PFE has to send traffic to another PFE the L2M header includes the fabric Stream ID (the Fabric Queue) of the destination PFE. So the LU chip of the ingress PFE (PFE 47) tells, via the L2M header, the MQ/XM chip to forward traffic to PFE 45. Then the PFE mask that is embedded in the fabric header will tell PFE 45 to forward the stream to 44. Before that, PFE 45 will reset bit 44 in the PFE mask and so on. Don't forget that each PFE also replicates traffic for its local interfaces with downstream multicast receivers.

Figure 5.4 summarizes the multicast forwarding path in the topology shown at the start of this chapter in Figure 5.1.

To finish our analysis let's use a previous command to check the fabric streams hop-by-hop (PFE by PFE) to retrieve the above multicast path.

PFE 47, 45, and 44 are located on the same MPC (in slot 11). Respectively, these PFEs on MPC 11 have a local PFE ID: 3, 1, and 0.

First check on the ingress PFE ID 3 on MPC 11 (global PFE ID 47), the statistics of fabric queue 45 (the low priority queue to reach PFE 45):

```
NPC11(R2 vty)# show cos halp fabric queue-stats 45
 [...]
PFE index: 3 CCHIP 0 Low prio Queue: 45 <<< From PFE 47 ( local ID 3) to PFE 45
Queued                :
  Packets             :      404706332          20007 pps
  Bytes               :   219755538276       10864096 Bps
Transmitted           :
  Packets             :      404706332          20007 pps   <<< copy sent
  Bytes               :   219755538276       10864096 Bps
  Tail-dropped pkts   :              0              0 pps
  Tail-dropped bytes  :              0              0 Bps
[...]
```

Great! Ingress PFE 47 sends a copy to PFE 45. Now let's execute the same command on the same MPC but change the destination PFE to 44 and from the point of view of source local PFE ID 1 (equal to the global PFE ID 45):

```
NPC11(R2 vty)# show cos halp fabric queue-stats 44
 [...]
PFE index: 1 CCHIP 0 Low prio Queue: 44 <<< From PFE 45 ( local ID 1) to PFE 44
Queued               :
  Packets            :      410381036          20000 pps
  Bytes              :   221195378404       10780176 Bps
Transmitted          :
  Packets            :      410381036          20000 pps <<< copy sent
  Bytes              :   221195378404       10780176 Bps
  Tail-dropped pkts  :              0              0 pps
[...]
```

PFE 45, which receives a copy from PFE 47, also sends a copy to PFE 44. Now call the same command on the same MPC but change the destination PFE to 0, and have a look at local PFE ID 0 (equal to the global PFE ID 44):

```
NPC11(R2 vty)# show cos halp fabric queue-stats 0
[...]
PFE index: 0 CCHIP 0 Low prio Queue: 0 <<< From PFE 44 ( local ID 0) to PFE 0
Queued               :
  Packets            :      672182703          20000 pps
  Bytes              :   359386529951       10700048 Bps
Transmitted          :
  Packets            :      672182703          20000 pps <<< copy sent
  Bytes              :   359386529951       10700048 Bps
  Tail-dropped pkts  :              0              0 pps
  Tail-dropped bytes:               0              0 Bps
[...]
```

The PFE 44, which receives a copy from PFE 45, also sends a copy to PFE 0, which is the last PFE in the PFE List. Great, you have followed the unary tree replication tree.

Remember: At each "PFE lookup" (of course, performed by LU) the PFE list is updated and the analysis of the composite next-host also conveyed in the fabric header helps each PFE to know if there are local replications to do.

Now let's suppose the multicast distribution tree is updated; for example, the interface xe-11/1/0 leaves the tree because the directly attached receiver sends an IGMP leave. The routing protocol daemon assigns a new multicast indirect next hop referring to a new OIL combination (plus a new composite next hop) and then updates the ingress PFE. This one recomputes on-the-fly the new PFE Mask used in the fabric header and switches on-the-fly to the new unary tree. All these tasks are carried out in a "make before break" way.

A similar mechanism is applied when the multicast distribution tree changes due to the power failure of a MPC that is in the middle of the replication tree. The PFE Liveness mechanism (also enabled with the enhanced-ip mode) allows fast PFE failure detection, as well as PFE-initiated replication tree recovery.

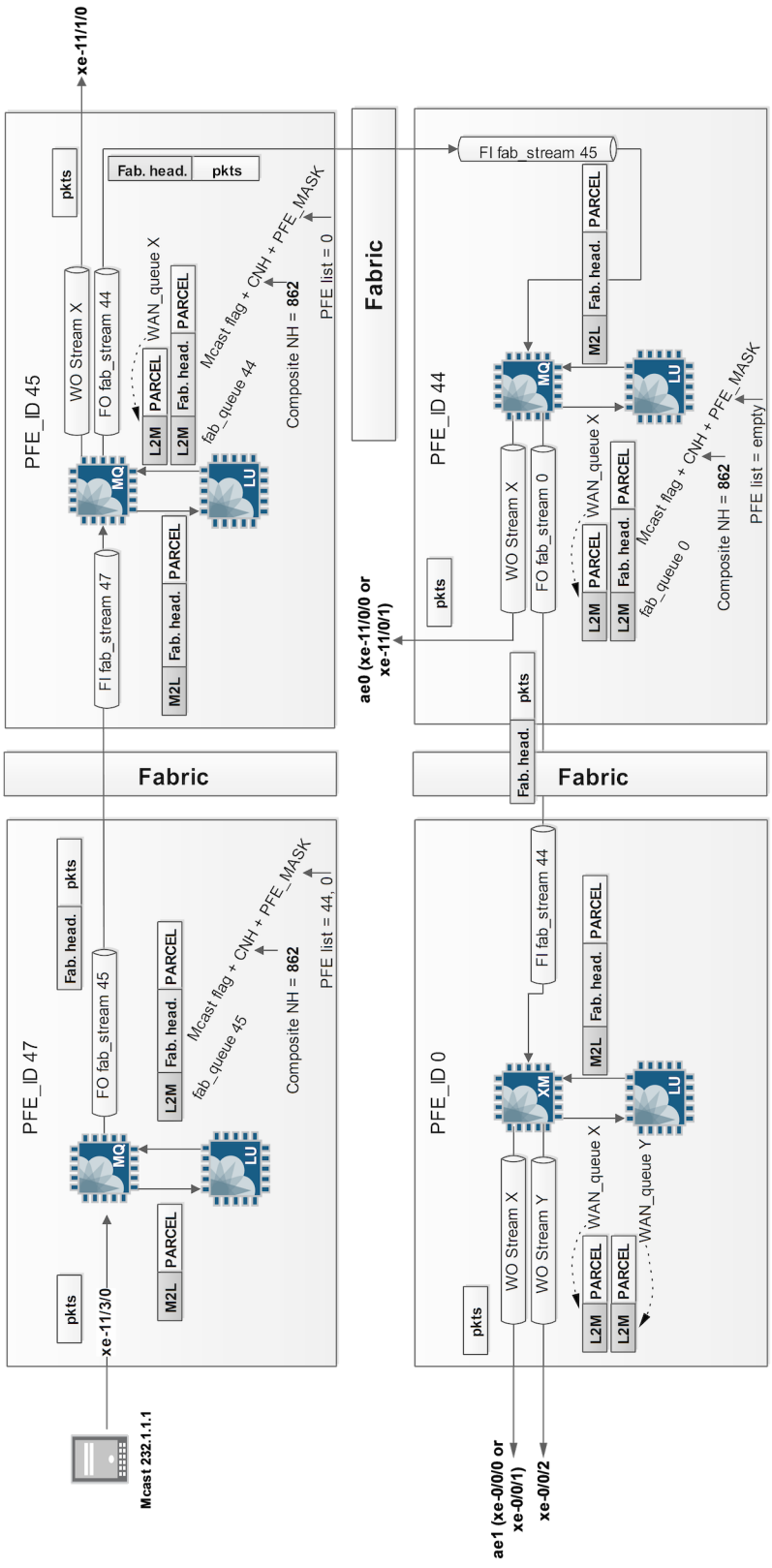



Figure 5.4 The Unary Tree Example

## Kernel Resolution

This is the last point to cover regarding multicast replication. *Multicast kernel resolution* means that the first multicast packets of a flow are punted to the RE because there is no state at the FIB level. The Routing daemon (RPD) then performs a multicast lookup (based on PIM join states) to reallocate a multicast indirect next hop and update the FIB.

Multicast Kernel Resolution is quite complex and is triggered in several cases:

- ASM mode. ASM multicast forwarding entries must be refreshed with the data plane. If a multicast stream stops but multicast control plane is still up (PIM joins) the multicast forwarding entry is removed from the RIB and FIB after 360 seconds. The next packet of the flow would need to be resolved by the RE's kernel.
- Sometimes when the incoming interface changes the multicast will be received from a new RPF neighbor.
- Multicast attack, exotic multicast configurations or simply misconfiguration.

Let's have a look at the default multicast routes in inet.1 table:

```
user@R2> show route table inet.1

inet.1: 3 destinations, 3 routes (3 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

224.0.0.0/4        *[Multicast/180] 5d 23:41:36
                       MultiResolve
232.0.0.0/8        *[Multicast/180] 5d 23:41:36
                       MultiResolve
```

As you can see, when there is no specific FIB entry for a given multicast stream, one of the default multicast routes (the ASM or SSM default route) is matched and the next hop associated is "Multicast Resolve."

To illustrate the kernel resolution mechanism let's take back the multicast topology, but stop the multicast receivers and the sender as well. Now, there are no more PIM states on R2:

```
user@R2> show pim join
Instance: PIM.master Family: INET
R = Rendezvous Point Tree, S = Sparse, W = Wildcard
```

Now, move to MPC 11 and display the route for the multicast stream (232.1.1.1;192.168.1.1):

```
NPC11(R2 vty)# show route long_ip prefix 232.1.1.1.192.168.1.1
IPv4 Route Table 6, R2/default.6, 0x0:
Destination                     NH IP Addr      Type     NH ID Interface
------------------------------- --------------- -------- ----- ---------
232/8                                            Resolve   851 RT-ifl 0 .local..6 ifl 332
```

The route lookup gives the "resolve" next hop as result. It means that the ingress LU chip that would receive this stream should "mark" the packet as exception and punt it to the RE for a kernel lookup (kernel resolution).

Only the first packets of each flow are sent up to the MPC microkernel CPU, which generates a resolve request and punts it to the RE in order to trigger the FIB update. Once the FIB update is pushed down to the LU chip, the packets of the stream are no longer punted. As mentioned, a Kernel Resolution is a type of exception. You can see the statistics of this exception by calling the following command. Here you see the statistics for PFE 3 of MPC 11, which hosts xe-11/3/0, the interface connected to the sender. Initially, before the source starts:

```
NPC11(R2 vty)# show jnh 3 exceptions

Reason                              Type           Packets       Bytes
================================================================
[...]

Routing
----------------------
resolve route                       PUNT(33)             0         0
```

As with any other exception, multicast resolution is rate-limited by DDoS-protection and queued to the Host Stream 1151 (in queue 6):

```
NPC11(R2 vty)# show  ddos policer resolve configuration
DDOS Policer Configuration:

                                          UKERN-Config   PFE-Config

 idx prot        group         proto on Pri  rate burst   rate burst
 ---  --- ------------ ------------ -- -- ------ ----- ------ -----
   1  100      resolve    aggregate  Y Md   5000 10000   5000 10000
   2  101      resolve        other  Y Lo   2000  2000   2000  2000
   3  102      resolve     ucast-v4  Y Lo   3000  5000   3000  5000
   4  103      resolve     mcast-v4  Y Lo   3000  5000   3000  5000
   5  104      resolve     ucast-v6  Y Lo   3000  5000   3000  5000
   6  105      resolve     mcast-v6  Y Lo   3000  5000   3000  5000

NPC11(R2 vty)# show ddos asic punt-proto-maps
PUNT exceptions directly mapped to DDOS proto:
code PUNT name                      group proto         idx q# bwidth  burst
---- --------------------      --------- ------       ---- -- ------ ------
[...]
                                  resolve mcast-v4      103  6   3000   5000
[...]
```

But you will see later that for the specific case of multicast resolution there is another specific default rate-limiter applied at the microkernel level (later in the exception path).

You can also retrieve the kernel resolve statistics, per incoming interface, via the CLI command:

```
user@R2> show multicast statistics
Interface: xe-11/3/0.0
    Routing protocol:          PIM   Mismatch error:              0
    Mismatch:                    0   Mismatch no route:           0
    Kernel resolve:              0   Routing notify:              0
    Resolve no route:            0   Resolve error:               0
    Resolve filtered:            0   Notify filtered:             0
    In kbytes:                   0   In packets:                  0
    Out kbytes:                  0   Out packets:                 0
Interface: xe-11/1/0.0
[...]
```

Now let's start the sender but keep the receivers down and check if there is a kernel resolution exception:

```
NPC11(R2 vty)# show jnh 3 exceptions terse

Reason                              Type           Packets      Bytes
=====================================================================

Routing
----------------------
resolve route                       PUNT(33)           450     233550
[...]
```

As you can see there are 450 packets that have been punted to the microkernel (which in turn punts a resolve request to the RE). This is due to the high rate of the multicast flow (20Kpps). Indeed, the kernel resolution may take some milliseconds.

Let's check back on multicast statistics at the RE level. You see only one kernel resolve for xe-11/3/0 interface. Actually the linecard CPU "forwards" only one resolve packet to the RE's kernel. This is all it needs to trigger a FIB update.

```
user@R2> show multicast statistics
[...]
Interface: xe-11/3/0.0
    Routing protocol:          PIM   Mismatch error:              0
    Mismatch:                    0   Mismatch no route:           0
    Kernel resolve:              1   Routing notify:              0
    Resolve no route:            0   Resolve error:               0
    Resolve filtered:            0   Notify filtered:             0
    In kbytes:                3543   In packets:                  5
    Out kbytes:                  0   Out packets:                 0
```

In scaled multicast networks, kernel resolution might request higher RE CPU and internal bandwidth consumption. To avoid that, Junos throttles multicast resolution requests at the microkernel level. Each linecard is limited to 66 resolutions per second (this is not the DDoS-protection policer). The following line card shell command gives you this information:

```
NPC11(R2 vty)# show nhdb mcast resolve
Nexthop Info:
   ID      Type    Protocol    Resolve-Rate
-----  --------  ----------  ---------------
  735   Resolve        IPv4              66
  736   Resolve        IPv6              66
  737   Resolve        IPv4              66
  738   Resolve        IPv6              66
  848   Resolve        IPv4              66
  849   Resolve        IPv6              66
  850   Resolve        IPv4              66
  851   Resolve        IPv4              66  <<<< multiresolve NH of 232/8 route
  852   Resolve        IPv6              66
  853   Resolve        IPv4              66
  857   Resolve        IPv4              66
  858   Resolve        IPv6              66
  859   Resolve        IPv4              66

NPC11(R2 vty)# show route long_ip prefix 232/8
IPv4 Route Table 6, R2/default.6, 0x0:
Destination                         NH IP Addr      Type     NH ID Interface
----------------------------------- --------------- -------- ----- ---------
232/8                                               Resolve   851 RT-ifl 0 .local..6 ifl 332
```

Now let's have a look at the multicast forwarding cache:

```
user@R2> show multicast route extensive
Group: 232.1.1.1
    Source: 192.168.1.1/32
    Upstream interface: xe-11/3/0.0
    Number of outgoing interfaces: 0
    Session description: Source specific multicast
    Statistics: 10380 kBps, 20000 pps, 2198494 packets
    Next-hop ID: 0
    Upstream protocol: PIM
    Route state: Active
    Forwarding state: Pruned
    Cache lifetime/timeout: 360 seconds
    Wrong incoming interface notifications: 0
    Uptime: 00:01:50
```

Interesting, as there is no PIM Join state for this (S,G), and the router has triggered a PIM Prune message toward the source to stop multicast traffic. The upstream router has been configured not to take into account the PIM Prunes messages in order to *force* the traffic down (with static IGMP at the downstream interface). This is why you still see packet statistics incrementing. If you have a look at the next hop part you can see that there is no indirect next hop associated. Indeed, as there is no PIM Join entry for this (S,G) the OIL is empty. But does it mean that the RE is continuously flooded by a multicast stream? No. Just look at PFE level on the forwarding table for the 232.1.1.1.192.168.1.1 route:

```
NPC11(R2 vty)# show route long_ip prefix 232.1.1.1.192.168.1.1
Destination                       NH IP Addr      Type     NH ID Interface
--------------------------------- --------------- -------- ----- ---------
232.1.1.1.192.168.1.1/64                          mdiscard   686 RT-ifl 378 .local..6 ifl 332
```

Kernel resolution triggered a FIB update and associated the multicast route entry to a specific next hop: multicast discard. Indeed, as there is no PIM join state available for this (S,G) the Kernel first prunes the stream and secures the control plane with a multicast FIB entry, which is a kind of black hole for this specific (S,G). Now, the multicast stream is silently discarded at the PFE level. Check PFE statistics (these drops are counted as normal discards):

```
user@R2> show pfe statistics traffic fpc 11 | match "normal"
    Normal discard             :          10051828983
```

The last question is: can I do a DOS (Denial Of Service) to the RE's Kernel if I try to play with the multicast kernel resolution? In other words, if I send packets with random (S,G) fields will they match the default multicast route at any time?

Of course, the answer is *no*.

You've seen previously that multicast kernel resolution is rate-limited to 66 resolutions per second. This limit is a µKernel limit. It means that each MPC can request a maximum of 66 multicast kernel resolutions per second, and these must be for different (S, G) since only one resolve request per (S, G) is sent up to the RE.

To illustrate this behavior, let's connect a traffic generator to xe-11/3/0 and send random multicast packets at 5Kpps. Let's keep source address equal to 192.168.1.1 and only the multicast group changes randomly at each packet. In this case, you can expect that each packet needs kernel resolution.

Remember, kernel resolution is an exception. The LU chip DDOS rate-limiter should work first. Let's have a look at statistics for the "resolve" exception:

```
NPC11(R2 vty)# show ddos policer resolve stats terse
DDOS Policer Statistics:

                                                    arrival   pass  # of
idx prot      group       proto on     loc     pass    drop    rate   rate flows
--- ---  ---------  ---------- --  ------   ------  ------  ------ ------ -----
  1  100    resolve   aggregate  Y   UKERN    43138       0    3001   3001     0
                                    PFE-0        0       0       0      0     0
                                    PFE-1        0       0       0      0     0
                                    PFE-2        0       0       0      0     0
                                    PFE-3    43138       0    5000   3001     0
  4  103    resolve    mcast-v4  Y   UKERN    43138       0    3001   3001     0
                                    PFE-0        0       0       0      0     0
                                    PFE-1        0       0       0      0     0
                                    PFE-2        0       0       0      0     0
                                    PFE-3    43138   25465    5000   3001     0
```

As you can see, the PFE 3 of MPC 11 triggers 5000 pps of multicast resolutions, but this flow of exceptions is rate-limited at 3000 pps by the LU chip's DDOS-protection policer. To confirm that PFE 3 delivers these 3Kpps to µKernel, you can check Queue 6 of the host stream. (Base queue for Stream 1151 is equal to 1016 of MPC 16x10GE):

```
NPC11(R2 vty)# show mqchip 3 dstat stats 0 1022  (0 means Qsys 0 - 1022=1016+6)

  QSYS 0 QUEUE 1022 colormap 2 stats index 72:

                 Counter      Packets     Pkt Rate            Bytes    Byte Rate
----------------------- ------------ ------------ ---------------- ------------
  Forwarded (NoRule)                0            0                0            0
  Forwarded (Rule)             967709         3001        539908122      1674837
```

Great! Finally, to validate that only 66 Kernel resolutions per second reach the routing engine, call the following command twice:

```
user@R2> show multicast statistics | match "interface|Kernel"
Sep 10 11:45:17

Interface: xe-11/3/0.0
    Kernel resolve:          13660   Routing notify:               0

user@R2> show multicast statistics | match "interface|Kernel"
Sep 10 11:45:28

Interface: xe-11/3/0.0
    Kernel resolve:          14389   Routing notify:               0
```

So, in 11 seconds the routing engine performed 729 kernel resolutions (14389-13660 ), or 66 kernel resolution per second.

And so this ends the multicast chapter.

# Appendices

# Appendix A : MPC CoS Scheduling

## MPC CoS Overview

The MQ or XM chip manages two kinds of hardware queues and performs *classical* CoS (classical meaning non-hierarchical CoS – without the QX chip). Enhanced Queueing MPCs (with QX or XQ chip) are not covered here.

- WAN queue : Eight hardware queues per physical interface by default.
- Fabric queue: Two fabric queues per destination PFE (even if eight queues are pre-provisioned by destination PFE – currently only two are used).

The SCHED block of the MQ and XM chip supports two levels of CoS, and each level is made of nodes. A *leaf* node is finally connected to the hardware queues. The terminology is the following:

- L1 node (first level): The L1 node represents a physical entity, in this case, a physical interface (for WAN CoS, interface means IFD) or a destination PFE (for fabric CoS). This is the scheduler of the physical interface.
- L2 node (second level): The L2 node represents a logical entity of the physical interface (unit or IFL). It is attached to an L1 node. The default CoS configuration (that means without per-unit scheduling) L2 node, although attached to an L1 node, won't be used. You can say that the L2 node is a dummy node.
- Q node: The Q node is actually a physical queue attached to L2 node. By default each L2 node has eight child Q nodes.
- Moreover, the MQ or XM SCHED block hosts several queuing systems (Qsys). You have to keep in mind that two Qsys are used on MPCs 16x10GE and MPC4e:
- Qsys 0 manages the L1/L2/Q nodes for the Physical WAN Output Stream (and the host – See Chapter 3).
- Qsys 1 manages the L1/L2/Q nodes for the fabric queues/streams.

Figure A.1 provides a graphical representation of the nodes' hierarchy.

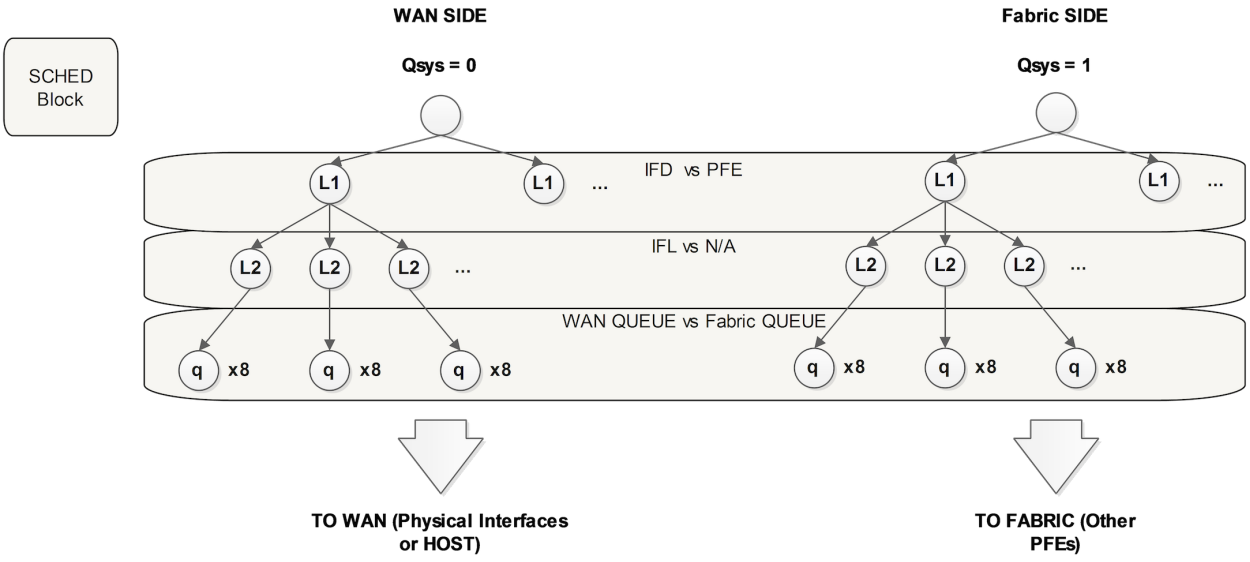



Figure A.1 The Scheduler Block of the MQ or XM Chip

Let's have a look at the entire CoS configuration on R2. The CoS configuration is quite atypical but it is only for troubleshooting and demonstration purposes. Four queues have been configured. For each, the priority, transmit-rate, and buffer-size are explicitly set. As you'll notice, the physical links are shaped to 555Mbits/s and one of the queues, the number 0, is further shaped at 222Mbits/s. Remember that a default inet-precedence BA classifier is used and only queue 3 has a fabric priority set to *high*:

```
user@R2> show configuration class-of-service
forwarding-classes {
    queue 0 FC0 priority low;
    queue 1 FC1 priority low;
    queue 2 FC2 priority low;
    queue 3 FC3 priority high;
}
interfaces {
    ae0 {
        scheduler-map my-sched;
        shaping-rate 555m;
        member-link-scheduler replicate;
    }
    ae1 {
        scheduler-map my-sched;
        shaping-rate 555m;
        member-link-scheduler replicate;
    }
}
scheduler-maps {
    my-sched {
        forwarding-class FC0 scheduler FC0-sched;
        forwarding-class FC1 scheduler FC1-sched;
        forwarding-class FC2 scheduler FC2-sched;
        forwarding-class FC3 scheduler FC3-sched;
    }
}
schedulers {
    FC0-sched {
        transmit-rate percent 50;
        shaping-rate 222m;
        buffer-size percent 50;
        priority low;
    }
    FC1-sched {
        transmit-rate percent 20;
        buffer-size percent 20;
        priority low;
    }
    FC2-sched {
        transmit-rate percent 20;
        buffer-size percent 20;
        priority low;
    }
    FC3-sched {
        transmit-rate percent 10;
        buffer-size percent 10;
        priority strict-high;
    }
}
```

With this configuration, the first four Forwarding Classes reserve the whole bandwidth, and nothing is left for the higher-numbered queues. Let's review how this CoS configuration is programed at the PFEs on the MPC.

## WAN CoS

To start the analysis on the WAN side, first, you need to find the associated L1 node / L2 node / Q nodes for a given physical interface (IFD). Let's arbitrarily choose one child link of AE0: the xe-11/0/1 interface (the forwarding next hop of Flow 2).

A single PFE command, available for both types of cards, is very useful to check CoS configuration at the PFE level and retrieve some of the SCHED block's information. To do that you need to find the IFD (interface device) index of your physical interface with this CLI command:

```
user@R2> show interfaces xe-11/0/1 | match "interface index"
  Interface index: 489, SNMP ifIndex: 727
```

Then use this PFE command on the MPC that hosts the IFD:

```
NPC11(R2 vty)# show cos halp ifd 489 1 <<< 1 means Egress direction
IFD name: xe-11/0/1   (Index 489)
    MQ Chip id: 0
    MQ Chip Scheduler: 0  <<<<< Qsys index
    MQ Chip L1 index: 32  <<<<< Our L1 node Index
    MQ Chip dummy L2 index: 64  <<<<< L2 node index is Dummy
    MQ Chip base Q index: 256   <<<<< Our Base Q node number
    Number of queues: 8
    Rich queuing support: 1 (ifl queued:0)
Queue    State         Max       Guaranteed   Burst Weight Priorities Drop-Rules
Index                  rate         rate      size           G    E   Wred  Tail
------ ----------- ------------ ------------ ------ ------ ---------- ----------
   256  Configured    222000000    222000000  32767     62   GL   EL     4   145
   257  Configured    555000000    111000000  32767     25   GL   EL     4   124
   258  Configured    555000000    111000000  32767     25   GL   EL     4   124
   259  Configured    555000000     Disabled  32767     12   GH   EH     4    78
   260  Configured    555000000            0  32767      1   GL   EL     0     7
   261  Configured    555000000            0  32767      1   GL   EL     0     7
   262  Configured    555000000            0  32767      1   GL   EL     0     7
   263  Configured    555000000            0  32767      1   GL   EL     0     7
------------------------------------------------------------------------------
```

You can see that in the CoS configuration at the PFE level, the Max rate column is actually the configured shaping-rate. Queue 0 shaping-rate configuration preempts the physical interface shaping-rate, this is why you see 222M as Max rate for Queue 0 instead of 555M. The Guaranteed rate is the per queue "reserved" bandwidth of the physical link. Since there is a shaping-rate at IFD level, the reference bandwidth is 555M and no longer 10G.

NOTE The calculated guaranteed rate is equal to (Transmit-rate % x physical link bandwidth / 100).

Figure A.2 gives you a graphical view of the nodes' hierarchy for the xe-11/0/1 interface.

In Figure A.2, the WAN CoS is managed by Qsys 0; the L1 node 32 is the scheduler of the physical interface with IFD 489 and finally the eight q nodes (from 256, the base queue, to 263) represent the physical hardware queues (each mapped from a forwarding class). L2 node is a dummy node.

To better understand the concepts of the SCHED block, let's try to analyze the hierarchy node-by-node and their configuration.

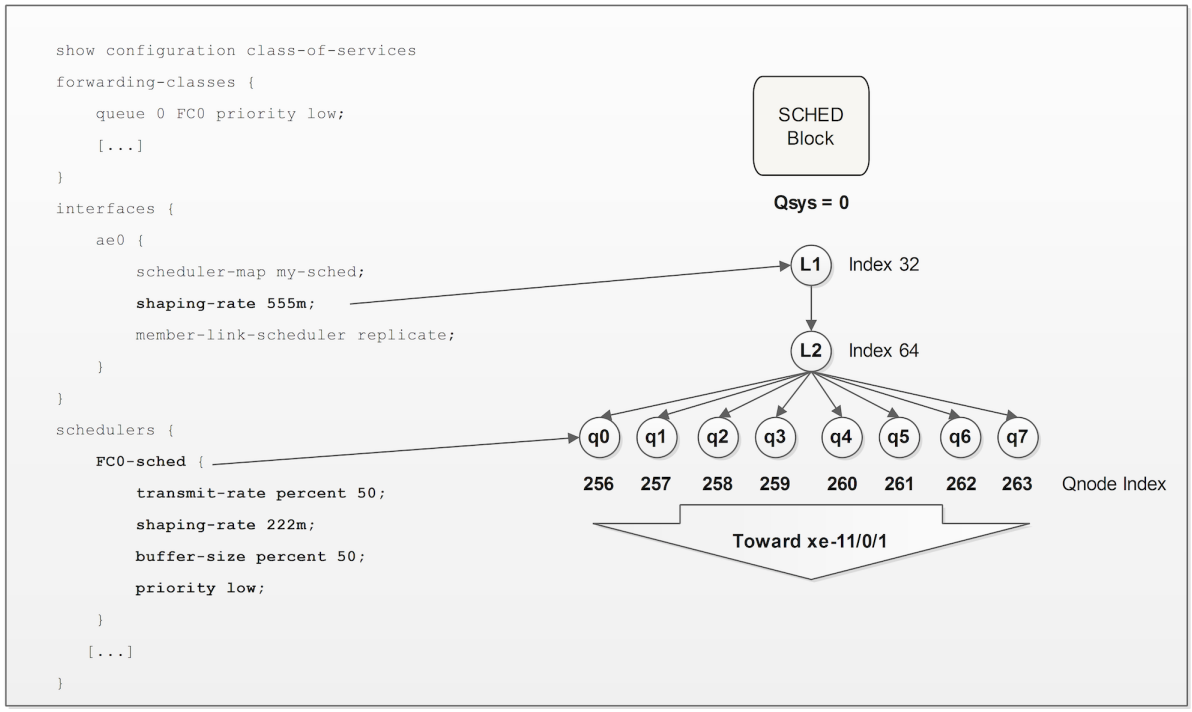



Figure A.2 The Scheduler Nodes Hierarchy for xe-11/0/1

NOTE The following is another method to check how CoS is programmed at the PFE level. If you prefer, use the previous PFE command (`show cos halp`) to globally retrieve CoS and SCHED block parameters.

First, let's try to find the L1 node assigned to xe-11/0/1 using two commands. Retrieve the Output Stream number of the xe-11/0/1 interface:

```
NPC11(R2 vty)# show mqchip 0 ifd
  [...]

  Output   IFD     IFD             Base
  Stream  Index    Name     Qsys   Qnum
  ------ ------ ---------- ------ ------
    1024    488  xe-11/0/0    MQ0      0
    1025    489  xe-11/0/1    MQ0    256  <<<< 1025 is the Wan Output Stream
    1026    490  xe-11/0/2    MQ0    512
    1027    491  xe-11/0/3    MQ0    776
```

And then call the second command to get the L1 node of xe-11/0/1 (Output Stream number 1025):

```
NPC11(R2 vty)# show mqchip 0 sched 0 l1-list

L1 nodes list for scheduler 0
-----------------------------
---------------------
L1 node    PHY stream
---------------------
0          1024
32         1025    <<<< L1 Node of xe-11/0/1 (Output Stream 1025)
64         1026
95         1119
```

```
96          1120
97          1027
127         1151
--------------------
```

tNOTE For XM-based cards like MPC4e, only one command is needed to get the L1 node of a given interface: `show xmchip <PFE-ID> ifd list 1` (the 1 means egress direction).

Okay, you've found that interface xe-11/0/1 is managed by L1 node 32 on the MQ Chip 0 of the MPC in slot 11. Let's see the L1 node #32 parameters and especially the configured physical shaping rate.

```
NPC11(R2 vty)# show mqchip 0 sched 0 l1 32
L1 node 32:
  allocated           : true
  child node start    : 64
  child node end      : 86
  rate enable         : 1
    m rate            : 555000000 bps <<<< The physical shaping rate
    m credits         : 2220
[...]
```

The sched 0 means Qsys 0, which is the queueing sytem that manages the WAN. L1 #32 is the L1 node index 32.

NOTE For XM-based cards like MPC4e a similar command is: `show xmchip <pfe-id> l1-node 0 <L1-Index>` (the first 0 means Qsys 0 and <L1-index> means L1 node index).

Now, let's see the child (L2 nodes) of L1 node #32:

```
NPC11(R2 vty)# show mqchip 0 sched 0 l1-children 32

Children for L1 scheduler node 32
---------------------------------

--------------------------------
L2 node    L1 node    PHY stream
--------------------------------
64         32         1025
65         32         1025
66         32         1025
67         32         1025
68         32         1025
69         32         1025
[...]
```

NOTE For XM-based card likes MPC4e a similar command is: `show xmchip <PFE-ID> l1-node children 0 <L1-Index>`.

As you can see, L1 node #32 has many children which are in the L2 nodes index. Remember, L2 node 64 (attached to IFD 489 – WAN Output Stream 1025) is a dummy node. Let's find out which Q nodes are attached to the L2 node:

```
NPC11(R2 vty)# show mqchip 0 sched 0 l2-children 64

Children for L2 scheduler node 64
---------------------------------

-------------------------------------------
Queue node  L2 node    L1 node    PHY stream
-------------------------------------------
256         64         32         1025 << Queue 0 in CoS configuration (FC0)
257         64         32         1025 << Queue 1 in CoS configuration (FC1)
```

```
258         64          32          1025 << Queue 2 in CoS configuration (FC2)
259         64          32          1025 << Queue 3 in CoS configuration (FC3)
260         64          32          1025
261         64          32          1025
262         64          32          1025
263         64          32          1025
```

You can see that L2 node #64 is connected to 8 Q nodes. Each of them is actually one physical hardware queue mapped sequentially to the queues and forwarding classes configured on the routers.

NOTE For XM-based cards like MPC4e a similar command is: `show xmchip <PFE-ID> l2-node children 0 <L2-Index>` (the first 0 means Qsys 0 and <L2-Index> is L2 node index).

Finally, let's do a final check of the Q node #256 configuration (which is actually the Queue 0 of the xe-11/0/1 interface), checking on the shaping rate assigned to this specific queue by our CoS configuration:

```
NPC11(R2 vty)# show mqchip 0 sched 0 q 256
Q node 256:
  allocated           : true
  parent node         : 64
  guarantee prio      : 3 GL
  excess prio         : 2 EL
  rate enable         : 1
    m rate            : 222000000 bps <<<<< Our Queue shaping rate is good. programmed
    m credits         : 888
  guarantee enable    : 1
    g rate            : 222000000 bps
    g credits         : 888
[...]
```

NOTE For XM-based cards like MPC4e a similar command is: `show xmchip <PFE-ID> q-node 0 <Q-Index>` (the first 0 means Qsys 0 and <Q-Index> means Q node index).

You can then retrieve statistics of a given Q node. The result is similar to the show interface queue CLI command, which shows all the statistics for all the queues assigned to a given interface. Here you just want to check the statistics of q-node #259, which is actually the Queue #3 of interface xe-11/0/1. Remember Flow 2 is classified into FC3:

```
NPC11(R2 vty)# show mqchip 0 dstat stats 0 259  (<<<< 0 259 means <Qsys> <Q-node>)

  QSYS 0 QUEUE 259 colormap 2 stats index 4452:

                  Counter          Packets     Pkt Rate            Bytes    Byte Rate
  ----------------------- ---------------- ------------ ---------------- ------------
    Forwarded (NoRule)                   0            0                0            0
    Forwarded (Rule)             345483936         1000     183773646744       532266
    Color 0 Dropped (WRED)               0            0                0            0
    Color 0 Dropped (TAIL)               0            0                0            0
    Color 1 Dropped (WRED)               0            0                0            0
    Color 1 Dropped (TAIL)               0            0                0            0
    Color 2 Dropped (WRED)               0            0                0            0
    Color 2 Dropped (TAIL)               0            0                0            0
    Color 3 Dropped (WRED)               0            0                0            0
    Color 3 Dropped (TAIL)               0            0                0            0
    Dropped (Force)                      0            0                0            0
    Dropped (Error)                      0            0                0            0

  Queue inst depth     : 0
  Queue avg len (taql): 0
```

As you can see, the 1000 pps of Flow 2 are queued in q-node 259 of (MPC 11, MQ chip 0), which is actually the Queue 3 of xe-11/0/1 interface.

NOTE For XM-based cards like MPC4e, a similar command is: `show xmchip <PFE-ID> q-node stats 0 <Q-node>` (the first 0 means Qsys 0 and **<Q-node>** means q-node index).

## Fabric CoS

You've already seen that fabric CoS is quite simple. Each PFE maintains two queues per destination PFE. One conveys low priority traffic and other high priority traffic. In the case of fabric congestion, it means: *when a PFE receives more traffic than the fabric bandwidth available between the PFE and the fabric planes, the low priority traffic will be dropped before the high priority traffic (on source PFEs)*. On the other hand, when the congestion point is the egress (destination) PFE, fabric drops occur via a back-pressure mechanism. In fact, when a destination PFE is oversubscribed, it does not allocate "credits" to a source PFE which tries to send packets to it, triggering packet buffering on the source PFE and ultimately drops when some packets reach the tail of the fabric queues.

Let's bring back Figure 2.6 from Chapter 2 that shows the Fabric Stream mapping of our case study :

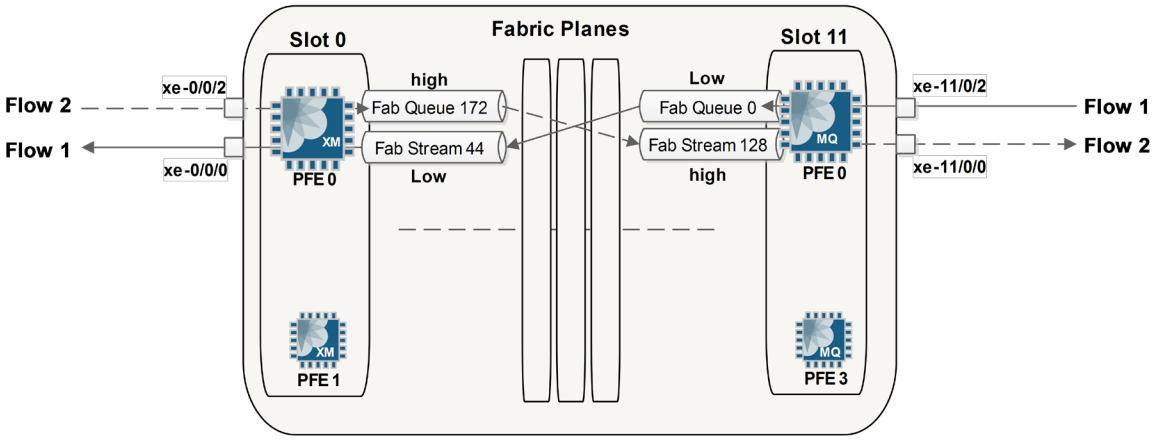



Figure A.3 Fabric Queues mapping for Flow 1 and 2

Even if the CLI command **`show class-of-service fabric statistics`** can give you fabric statistics per (source, destination) MPC pair, sometimes you may need to have a more granular view and retrieve per (source, destination) PFE pair traffic statistics. Again, a nice PFE command can help you. The command takes as argument the global destination PFE number ( = MPC_Destination_Slot * 4 + PFE_ID):

```
NPC11(R2 vty)# show cos halp fabric queue-stats 0 (<< 0 means The PFE 0 in slot 0)
PFE index: 0 CChip 0 Low prio Queue: 0
Queued                :
  Packets             :    29724223195          1001 pps
  Bytes               : 16110235432270        510064 Bps
Transmitted           :
  Packets             :    29724223195          1001 pps
  Bytes               : 16110235432270        510064 Bps
  Tail-dropped pkts   :              0             0 pps
  Tail-dropped bytes  :              0             0 Bps
  RED-dropped pkts    :
```

```
    Low             :              0              0 pps
    Medium-low      :              0              0 pps
    Medium-high     :              0              0 pps
    High            :              0              0 pps
  RED-dropped bytes :
    Low             :              0              0 Bps
    Medium-low      :              0              0 Bps
    Medium-high     :              0              0 Bps
    High            :              0              0 Bps
  RL-dropped pkts   :              0              0 pps
  RL-dropped bytes  :              0              0 Bps

PFE index: 0 CChip 0 High prio Queue: 128
Queued              :
  Packets           :         283097              0 pps
  Bytes             :      144250534              0 Bps
[...]
PFE index: 1 CChip 0 Low prio Queue: 0
[...]
PFE index: 1 CChip 0 High prio Queue: 128
[...]
PFE index: 2 CChip 0 Low prio Queue: 0
[...]
PFE index: 2 CChip 0 High prio Queue: 128
[...]
PFE index: 3 CChip 0 Low prio Queue: 0
[...]
PFE index: 3 CChip 0 High prio Queue: 128
[...]
```

How to interpret this output? This command gives you the fabric statistics of two (low, high) fabric queue/streams towards a given destination PFE (the argument of the command), from each source PFE attached to the MPC on which the command is executed. Here, you are attached to MPC in slot 11, which receives Flow 1 and forwards it to the MPC in slot 0 PFE_ID 0. And this is why you check the fabric statistics for the global PFE_ID 0. Following the configuration, Flow 1 is received on xe-11/0/2 attached to PFE 0, and is marked as low priority traffic. PFE 0 of MPC slot 11 has two fabric queues for Destination PFE 0 of MPC 0: Fabric queue (Stream) 0 for low priority traffic; and Fabric queue (Stream) 128 for high priority traffic. Here, you have only packets queued on fabric queue number 0 (Low Priority).

Let's use the same command on MPC 0 for global destination PFE 44 (PFE 0 of MPC 11):

```
NPC0(R2 vty)# show cos halp fabric queue-stats 44

PFE index: 0 CChip 0 Low prio Queue: 44
[...]
PFE index: 0 CChip 0 High prio Queue: 172
Queued              :
  Packets           :      672109500          1000 pps
  Bytes             :   342775716064        509328 Bps
Transmitted         :
  Packets           :      672109500          1000 pps
  Bytes             :   342775716064        509328 Bps
  Tail-dropped pkts :              0             0 pps
  Tail-dropped bytes:              0             0 Bps
  RED-dropped pkts  :
    Low             :              0             0 pps
    Medium-low      :              0             0 pps
    Medium-high     :              0             0 pps
    High            :              0             0 pps
  RED-dropped bytes :
```

```
    Low             :              0              0 Bps
    Medium-low      :              0              0 Bps
    Medium-high     :              0              0 Bps
    High            :              0              0 Bps
  RL-dropped pkts   :              0              0 pps
  RL-dropped bytes  :              0              0 Bps
PFE index: 1 CChip 0 Low prio Queue: 44
[...]
PFE index: 1 CChip 0 High prio Queue: 172
[...]
```

Here you can see that Flow 2 of (MPC 0, PFE 0) is queued into fabric queue 172 of the XM chip 0. This queue is the fabric queue that handles high priority traffic that wants to reach the PFE 0 of MPC 11 (Global PFE_ID 44).

With line card shell CoS commands, you can retrieve these statistics by directly calling the q-node dedicated to a given fabric queue.

Let's quickly work through an example to cement your understanding of MQ/XM CoS. Remember, fabric CoS is managed by the Qsystem 1 of MQ or XM chip.

For example, on MPC slot 11, let's retrieve the statistics of q-node 0 (Qsys 1):

```
NPC11(R2 vty)# show mqchip 0 dstat stats 1 0 <<<< 1 0 means <Qsys> <q-node>

  QSYS 1 QUEUE 0 colormap 2 stats index 288:

                   Counter          Packets     Pkt Rate            Bytes    Byte Rate
  ------------------------ --------------- ------------ ---------------- ------------
    Forwarded (NoRule)                   0            0                0            0
    Forwarded (Rule)            29725737468         1000   16111006996450       510311
```

NOTE Fabric q-node is exactly the same value as fabric queue or fabric stream, so you can use the same rules explained in Figure 2.6.

And an example on MPC slot 0, would be to retrieve the statistics of q-node 172 (Qsys 1), mapped to the high priority stream that points to (MPC 11, PFE 0):

```
NPC0(R2 vty)# show xmchip 0 q-node stats 1 172  <<<< 1 172 means <Qsys> <q-node>
Queue statistics (Queue 0172)
-----------------------------

--------------------------------------------------------------------------------------------
Color  Outcome              Counter  Counter  Total                Rate
                            Index    Name
--------------------------------------------------------------------------------------------
All    Forwarded (No rule)  2160     Packets  0                    0 pps
All    Forwarded (No rule)  2160     Bytes    0                    0 bps
All    Forwarded (Rule)     2161     Packets  672871637            1000 pps
All    Forwarded (Rule)     2161     Bytes    343164405934         4080000 bps
```

This last command ends the CoS sub-discussion, a delightful tangent from our packet life analysis of the mighty MX Series 3D Router, where we were talking about the SCHED block.

# Appendix B: More on Host Protection

This appendix provides more insight on the host protection feature set. First, the DDoS protection built-in feature (already outlined in Chapter 3) is demonstrated with a case study based on ICMP.

Next, the ARP (Address Resolution Protocol) specific case is explained more in detail. ARP storms are a classical symptom of layer 2 loops, and the host protection logic in these particular cases involves more mechanisms, apart from DDoS protection.

In both case studies (ICMP and ARP), you will see that some additional protocol-specific policers are in place, and these policers do not belong to the DDOS Protection feature. The latter is a relatively recent feature, while the ICMP and ARP specific policers were implemented much earlier in Junos. Host protection is always a good thing, so all these security measures (older and newer) are left in place, and they are cascaded in a specific order.

CAUTION During the following two case studies, sometimes policers are removed or relaxed. This is shown for lab demonstration purposes only and it is definitely a bad practice in production networks.

## DDOS Protection Case Study

In order to see DDoS protection in action, let's see what happens when R2 receives more ICMP echo request packets that are allowed by the system. This should be a good practice to review some previous commands and can help you to troubleshoot the MX when it experiences DDOS attacks. The following case study will also help you to understand how drops are managed by the MPC.

Let's start by sending 200Kpps of ICMP echo requests from R4 towards R2, and the same rate of ICMP sent by R5 towards R2. The attacks are received by two interfaces on R2 not in LAG. Figure B.1 shows you the simple new topology to simulate our attack.

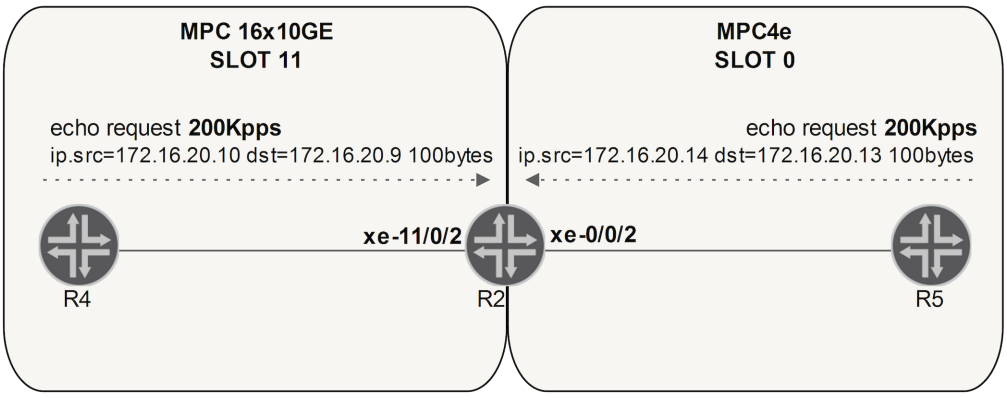



Figure B.1 Topology to Simulate a DDOS Attack

Remember, there are two mechanisms to protect the R2 router from this ICMP attack:

- The ICMP policer applied the input lo0.0 firewall filter. This policer acts in a per-PFE level basis and it deals with the host-inbound traffic arriving from all the interfaces at that PFE. Since each flow is arriving at a different PFE, the policer rate-limits each ICMP traffic flow to 100Mbits/s independently:

```
user@R2> show configuration
interfaces {
    lo0 {
        unit 0 {
            family inet {
                filter {
                    input protect-re;
                }
                address 172.16.21.2/32 {
                    primary;
                }
            }
        }
    }
}
firewall {
    family inet {
        filter protect-re {
            term ICMP {
                from {
                    protocol icmp;
                }
                then {
                    policer ICMP-100M;
                    count ICMP-CPT;
                    accept;
                }
            }
            term OTHER {
                then accept;
            }
        }
    }
    policer ICMP-1M {
        if-exceeding {
            bandwidth-limit 100m;
            burst-size-limit 150k;
        }
        then discard;
    }
}
```

- The ICMP DDOS protection has been *explicitly configured* at `[edit system ddos-protection]` for a maximum of **500pps** (the default value is overridden). This enforces an input ICMP rate limit at three levels: the LU chip, the µKernel and the RE:

```
user@R2> show configuration
system {
    ddos-protection {
        protocols {
            icmp {
               aggregate {
                    bandwidth 500;
                    burst 500;
                }
            }
        }
    }
}
```

It's time to start the DDOS attack coming from R1 and R3. First you can check the incoming rate on the xe-11/0/2 and xe-0/0/2 interfaces and confirm that R2 receives both attacks:

```
user@R2> show interfaces xe-*/0/2 | match "Physical|rate"
Physical interface: xe-0/0/2, Enabled, Physical link is Up
  Input rate     : 131201448 bps (200002 pps)
  Output rate    : 0 bps (0 pps)
Physical interface: xe-11/0/2, Enabled, Physical link is Up
  Input rate     : 131199536 bps (199999 pps)
  Output rate    : 0 bps (0 pps)
```

The first rate limiting is done by the ICMP policer of our lo0.0 input firewall filter. This is done at the LU chip level. One could expect that the WI blocks of the (MPC 11, MQ chip 0) and (MPC 0, XM chip 0) still sees the 200Kpps, because WI is before LU.

Let's try to check this fact. First step, you need to retrieve the Physical Wan Input Stream associated to interfaces xe-11/0/2 and xe-0/0/2. Remember, ICMP traffic is conveyed in the CTRL Stream (or, the medium stream):

```
user@R2> request pfe execute target fpc11 command "show mqchip 0 ifd" | match xe-11/0/2 | trim 5
     1033    592  xe-11/0/2     66     hi
     1034    592  xe-11/0/2     66    med  <<<< CTRL WAN Input Stream is 1034
     1035    592  xe-11/0/2     66     lo
     1040    592  xe-11/0/2     66   drop
     1026    592  xe-11/0/2    MQ0     16

user@R2> request pfe execute target fpc0 command "show xmchip 0 ifd list 0" | match xe-
0/0/2 | trim 5
 xe-0/0/2           573          1033          66        0 (High)
 xe-0/0/2           573          1034          66        1 (Medium) <<<< CTRL WAN
Input Stream is 1034
 xe-0/0/2           573          1035          66        2 (Low)
 xe-0/0/2           573          1072          66        3 (Drop)
```

Then, enable WI accounting for this specific incoming WAN stream on both cards:

```
user@R2> request pfe execute target fpc11 command "test mqchip 0 counter wi_rx 0 1034"
user@R2> request pfe execute target fpc0 command " test xmchip 0 wi stats stream 0 1034"
```

And let's collect the statistics:

```
user@
R2> request pfe execute target fpc11 command "show mqchip 0 counters input stream 1034" | trim 5
WI Counters:

             Counter          Packets     Pkt Rate            Bytes    Byte Rate
  ------------------ --------------- ------------ ---------------- ------------
      RX Stream 1034 (010)  102815359       200029    <<<< At WI level we still see 200Kpps
[…]

user@R2> request pfe execute target fpc0 command "show xmchip 0 phy-
stream stats 1034 0" | find "Tracked stream" | trim 5

Tracked stream statistics
-------------------------

Track Stream Stream Total Packets        Packets Rate
-------------------------------------------------------------------------------
0     0x7f   0xa    26081598            199998  <<<< At WI level we still see  200Kpps
[...]
```

So at the WI level there are still 200Kpps. Let's check the lo0.0 ICMP policer, but first, don't forget to disable the WI accounting configuration:

```
user@R2> request pfe execute target fpc11 command "test mqchip 0 counter wi_rx 0 default"
user@R2> request pfe execute target fpc0 command " test xmchip 0 wi stats default 0 0"
```

## ICMP Policer at lo0.0 Input Filter

Now you can check that the lo0.0 ICMP policer works well (at the RE level) and on each PFE:

```
user@R2> show firewall filter protect-re

Filter: protect-re
Counters:
Name                                            Bytes              Packets
ICMP-CPT                                   65777268022            802161589
Policers:
Name                                            Bytes              Packets
ICMP-100M-ICMP                             20505314130            250064781

NPC11(R2 vty)# show filter
Term Filters:
------------
   Index    Semantic   Name
--------  ---------- ------
       8  Classic    protect-re

NPC11(R2 vty)# show filter index 8 counters  <<<< ON MPC 11
Filter Counters/Policers:
   Index          Packets                 Bytes  Name
--------  ---------------  --------------------  --------
       8        416615343           34162464430  ICMP-CPT
       8        129984451          10658727070  ICMP-100M-ICMP(out of spec)
       8                0                    0  ICMP-100M-ICMP(offered)
       8                0                    0  ICMP-100M-ICMP(transmitted)

NPC0(R2 vty)# show filter index 8 counters  <<<< And on MPC 0
Filter Counters/Policers:
   Index          Packets                 Bytes  Name
--------  ---------------  --------------------  --------
       8        432288418           35447719532  ICMP-CPT
       8        134644664          11040862448  ICMP-100M-ICMP(out of spec)
       8                0                    0  ICMP-100M-ICMP(offered)
       8                0                    0  ICMP-100M-ICMP(transmitted)
```

## DDOS Protection at the LU level

This attack is not entirely absorbed by the firewall filter ICMP Policer (100Mbits/s are allowed). This is the job of the DDOS protection feature. Let's have a look at the DDOS violation logs:

```
user@R2> show ddos-protection protocols violations
Packet types: 198, Currently violated: 1

Protocol    Packet      Bandwidth  Arrival   Peak      Policer bandwidth
group       type        (pps)      rate(pps) rate(pps) violation detected at
icmp        aggregate   500        305355    305355    2014-05-28 10:20:55 CEST
          Detected on: RE, FPC-0, 11
```

Why don't we see 400Kpps at arrival rate but rather 305Kpps in the show output? This is due to the ICMP policer of the lo0 that was discarded before a part of this attack (between 100Kpps – 50Kpps per attack).

Now, let's have a look at the DDOS statistics on the RE and on each MPC:

```
user@R2> show ddos-protection protocols icmp statistics
Packet types: 1, Received traffic: 1, Currently violated: 1
Protocol Group: ICMP
[…]

 Routing Engine information:
      Aggregate policer is currently being violated!
        Violation first detected at: 2014-05-28 10:20:55 CEST
        Violation last seen at:      2014-05-28 10:33:26 CEST
        Duration of violation: 00:12:31 Number of violations: 3
      Received:  4813886             Arrival rate:     1000 pps
      Dropped:   1725094             Max arrival rate: 1002 pps
        Dropped by individual policers: 0
        Dropped by aggregate policer:   1725094

    FPC slot 0 information:
      Aggregate policer is currently being violated!
        Violation first detected at: 2014-05-28 10:20:55 CEST
        Violation last seen at:      2014-05-28 10:33:26 CEST
        Duration of violation: 00:12:31 Number of violations: 3
      Received:  522942303           Arrival rate:     152520 pps
      Dropped:   520899678           Max arrival rate: 200409 pps
        Dropped by individual policers: 0
        Dropped by aggregate policer:   520899678
        Dropped by flow suppression:    0

    FPC slot 11 information:
      Aggregate policer is currently being violated!
        Violation first detected at: 2014-05-28 10:20:57 CEST
        Violation last seen at:      2014-05-28 10:33:26 CEST
        Duration of violation: 00:12:29 Number of violations: 3
      Received:  527482854           Arrival rate:     152772 pps
      Dropped:   524714954           Max arrival rate: 200409 pps
        Dropped by individual policers: 0
        Dropped by aggregate policer:   524714954
        Dropped by flow suppression:    0
```

You can see that that on each MPC, the DDOS ICMP AG policer receives 150Kpps. The limit is configured explicitly to 500pps. So the LU chip will deliver only 500pps to the µKernel. As the attack is distributed over two MPCs, each MPC will deliver 500pps to the RE. This is why you can see 1000pps at the RE level. And again, the RE AG policer that also has a threshold of 500pps, only delivers 500pps to the system. Let's drill down with PFE commands to get a more detailed view. This example is on MPC 11:

```
user@R2> request pfe execute target fpc11 command "show ddos policer icmp stats" | trim 5
DDOS Policer Statistics:
arrival   pass  # of
idx prot   group       proto on     loc      pass       drop    rate   rate flows
---  ---  ------  ---------- --  ------  --------   --------  ------ ------ ----
 68  900    icmp   aggregate  Y   UKERN   3272837          0     499    499    0
                                  PFE-0   2745901  677333644  152442    499    0
                                  PFE-1         0          0       0      0    0
                                  PFE-2         0          0       0      0    0
                                  PFE-3         0          0       0      0    0
```

And you can see the rate in the output. Finally, in order to be sure that the HOST Stream (1151) conveys only the 499pps packets, you can use this set of commands that look at HOST Stream Queues:

```
user@R2> request pfe execute target fpc11 command "show cos halp queue-resource-
map 0" | match "stream-id|1151" | trim 5   <<< 0 means PFE_ID 0
  stream-id  L1-node  L2-node  base-Q-Node
       1151      127      254         1016  <<< 1016 is the base queue (queue 0)

user@R2> request pfe execute target fpc11 command "show mqchip 0 dstat stats 0 1016" | trim 5
  QSYS 0 QUEUE 1016 colormap 2 stats index 0:

                 Counter      Packets     Pkt Rate         Bytes    Byte Rate
----------------------- ------------ ------------ ------------- ------------
  Forwarded (NoRule)               0            0             0            0
  Forwarded (Rule)           4894829          499     598458981        60379
  Color 0 Dropped (WRED)           0            0             0            0
  Color 0 Dropped (TAIL)           0            0             0            0
  Color 1 Dropped (WRED)           0            0             0            0
  Color 1 Dropped (TAIL)           0            0             0            0
  Color 2 Dropped (WRED)           0            0             0            0
  Color 2 Dropped (TAIL)           0            0             0            0
  Color 3 Dropped (WRED)           0            0             0            0
  Color 3 Dropped (TAIL)           0            0             0            0
  Dropped (Force)                  0            0             0            0
  Dropped (Error)                  0            0             0            0
```

One last important thing to notice regarding drops is that the discards triggered by the firewall filter policer or by the DDOS policers are detected by the LU chip, but the drops are really taking place at the MQ/XM chip level. Indeed, when the LU chip detects that a packet (Parcel) is out of spec, it sends the Parcel back with a notification in the L2M header that the packet has to be dropped. These PFE drops are counted as normal discards:

```
user@R2> show pfe statistics traffic fpc 11 | match "Discard|drop"
    Software input control plane drops  :                    0
    Software input high drops           :                    0
    Software input medium drops         :                    0
    Software input low drops            :                    0
    Software output drops               :                    0
    Hardware input drops                :                    0
Packet Forwarding Engine hardware discard statistics:
    Timeout                     :                    0
    Normal discard              :           1786267748
    Extended discard            :                    0
    Info cell drops             :                    0
    Fabric drops                :                    0
```

The above command shows that there is no congestion on the Host inbound path and the control plane is well protected. Indeed, there are no hardware input drops (aggregated drop stats of the eight hardware queues of the stream 1151) and no software input drops (drop stats of the µKernel queue – towards the RE).

## The FreeBSD ICMP Policer

In order to observe congestion drops (just for fun), you can disable in the lab the ICMP DDOS protection and keep only the lo0.0 firewall filter. Let's try it:

```
user@R2# set system ddos-protection protocols icmp aggregate disable-fpc
user@R2# set system ddos-protection protocols icmp aggregate disable-routing-engine
```

New check for PFE Drops on one MPC:

```
user@R2> show pfe statistics traffic fpc 11 | match "Discard|drop"
    Software input control plane drops  :                     0
    Software input high drops           :                     0
    Software input medium drops         :                     0
    Software input low drops            :                     0
    Software output drops               :                     0
    Hardware input drops                :              51935961
Packet Forwarding Engine hardware discard statistics:
    Timeout                     :                   0
    Normal discard              :          1876852127
    Extended discard            :                   0
    Info cell drops             :                   0
```

You can now see hardware input drops, which means that the hardware queue(s) of the Host stream managed by the MQ/XM chip are congested. This can be confirmed by using the following command:

```
user@R2> request pfe execute target fpc11 command "show mqchip 0 dstat stats 0 1016" | trim 5

 QSYS 0 QUEUE 1016 colormap 2 stats index 0:
                   Counter      Packets     Pkt Rate        Bytes    Byte Rate
  ------------------------ ------------ ------------ ------------ ------------
    Forwarded (NoRule)                0            0            0            0
    Forwarded (Rule)             689014        15765     83370694      1907625
    Color 0 Dropped (WRED)      5926034       136629    717050114     16532169
    Color 0 Dropped (TAIL)          404            0        48884            0
    Color 1 Dropped (WRED)            0            0            0            0
    Color 1 Dropped (TAIL)            0            0            0            0
    Color 2 Dropped (WRED)            0            0            0            0
    Color 2 Dropped (TAIL)            0            0            0            0
    Color 3 Dropped (WRED)            0            0            0            0
    Color 3 Dropped (TAIL)            0            0            0            0
    Dropped (Force)                   0            0            0            0
    Dropped (Error)                   0            0            0            0
```

Indeed, Queue 0 (Absolute queue 1016 of Qsys 0) of the Host stream experienced RED and TAIL drops (remember this is due to back pressure from TOE). In this case the MPC 11 delivers "only" 15765pps to the µKernel. µKernel seems to accept these 15Kpps without any issue and this is why there are no Software Input drops with the `show pfe statistics traffic fpc 11` command. You can also call the `show ttp statistics` command on MPC 11 to double-check for software drops.

Nevertheless, 15kpps of ICMP per MPC doesn't mean that the RE will answer all the 30Kpps echo requests. Actually, the FreeBSD system itself has its own ICMP rate-limiter, which acts as yet another level of protection:

```
user@R2> start shell
% sysctl -a | grep icmp.token
net.inet.icmp.tokenrate: 1000
```

NOTE The FreeBSD ICMP policer that can be configured at [edit system internet-options icmpv4-rate-limit] hierarchy level. This policer has a default 1000pps value, and that is the reason why the ICMP DDOS policer was previously set to 500pps: the goal was to see DDOS Protection in action, and not the FreeBSD ICMP kernel policer.

OK, this is why you only see 500pps of ICMP echo replies per MPC and why you can notice a lot of rate-limited ICMP packets at the system Level:

```
user@R2> show pfe statistics traffic fpc 0
Packet Forwarding Engine traffic statistics:
    Input  packets:              31895070                199345 pps
    Output packets:                 79641                   500 pps

user@R2> show pfe statistics traffic fpc 11
Packet Forwarding Engine traffic statistics:
    Input  packets:              53421523                199337 pps
    Output packets:                132774                   500 pps

user@R2> show system statistics icmp | match "rate limit"
         100338743 drops due to rate limit
```

Let's configure back DDOS protection and stop the ICMP DDOS attack, take a break if necessary, and then let's move on to how ARP packets are handled by MX 3D router. It should be a very interesting walkthrough.

## Handling Inbound ARP Packets

This section of Appendix B covers some specific processing done for ARP (Address Resolution Protocol) packets. ARP packets are received and handled by the host, but here the focus is only on the input direction.

To illustrate ARP requests, let's go back to the same topology and force R1 and R3 to send an ARP request toward R2 every second, as shown in Figure B.2.

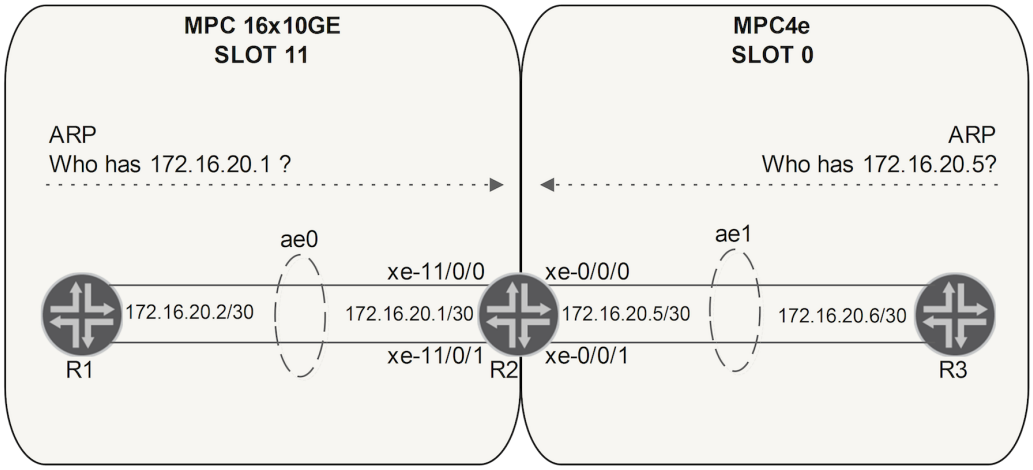



Figure B.2 The ARP Request Case

An ARP packet is a Layer 2 packet carried directly over Ethernet with the EtherType equal to 0x0806. As you've previously seen, packets coming into the MX MPC are first pre-classified and then handled by the MQ or XM chip. In the case of ARP packets, they are conveyed in the CRTL stream at the WI functional block level. The WI turns the ARP request into a Parcel (because the entire ARP packet is lower than 320 bytes) and sends it to the LU chip.

Figure B.3 illustrates the life of an ARP packet inside the PFE.

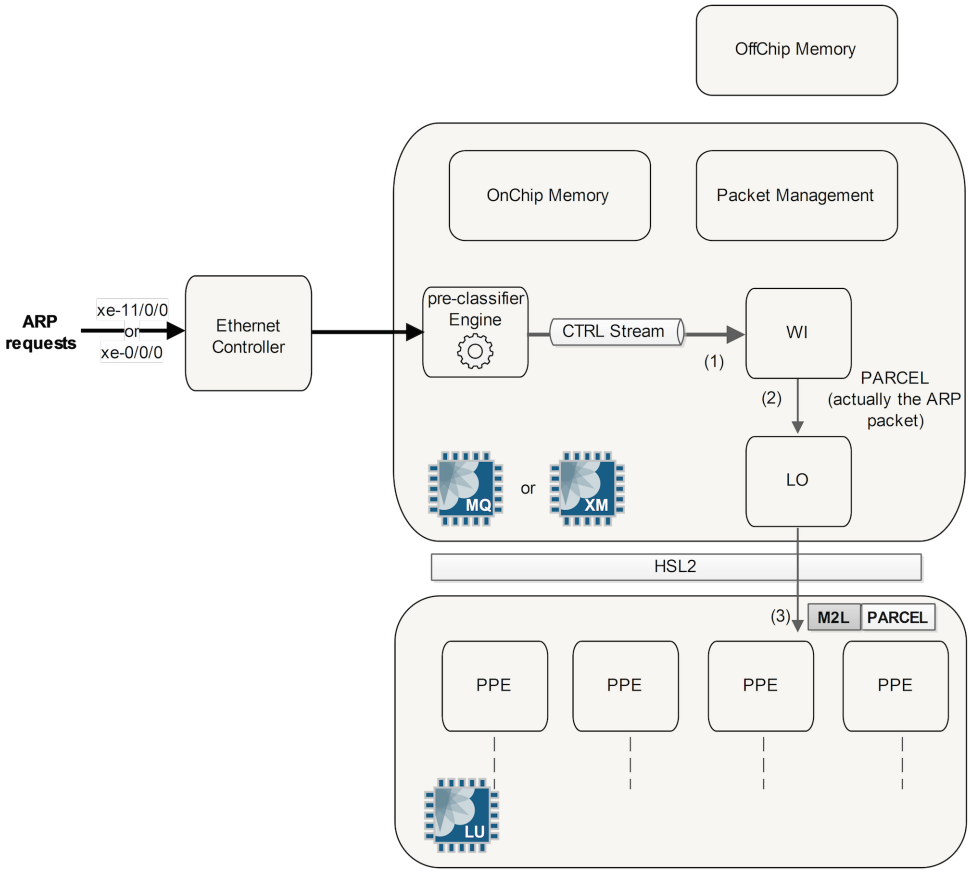



Figure B.3 How the MX 3D Handles ARP Requests

The LU chip then performs several tasks. First it performs an Ethernet frame analysis. By checking the EtherType the LU chip deduces that the packet is an ARP packet, so it flags it as an exception, more specifically, as a control packet (PUNT 34 exception code).

If you execute the PFE commands several times on MPC 11 or MPC 0, you will see the counter incrementing by one per second – the rate of the ARP streams from R1 or R3.

```
NPC11(R2 vty)# show jnh 0 exceptions terse
Reason                              Type         Packets       Bytes
==========================================================================
control pkt punt via nh             PUNT(34)         150        6962
```

NOTE Remember that in this topology there is no protocol enabled (not even LACP), this is why you only see 1pps on this counter.

As the ARP packet is a pure Layer 2 packet it will not be processed by the lo0.0 input firewall filter or by the HBC firewall filter, which are both Layer 3 family-based. Nevertheless, ARP can be policed by three types of policers:

- Default per-PFE ARP policer
- Configurable per-IFL (Interface Logical) ARP policer
- ARP DDOS (hierarchical) policer

These three policers are managed by the LU chip (and the DDOS policer is also applied at higher levels), but as dicussed previously, the real discard of packets that exceed the policer rate is performed by the the MQ/XM chip. The LU chip just sends back the information to the MQ/XM chip that the packet must be discarded.

## The Default per-PFE ARP Policer

The first policer is a non-configurable policer which is applied by default on ethernet interfaces as an input policer. This is a per-PFE ARP policer named __default_arp_policer__. To check if a given IFD has this policer applied, just call the following CLI command:

```
user@R2> show interfaces policers ae[0,1].0
Interface       Admin Link Proto Input Policer         Output Policer
ae0.0           up    up
                           inet
                           multiservice __default_arp_policer__
Interface       Admin Link Proto Input Policer         Output Policer
ae1.0           up    up
                           inet
                           multiservice __default_arp_policer__
```

NOTE You can disable the default per-PFE ARP policer (set interfaces <*> unit <*> family inet policer disable-arp-policer).

As you can see, both AE0 and AE1 have it enabled. But what is the value of this default ARP policer? To find out let's move back to the PFE and check the policer program:

```
NPC11(R2 vty)# show filter
Program Filters:
---------------
   Index     Dir     Cnt    Text     Bss  Name
--------  ------  ------  ------  ------  --------

Term Filters:
------------
   Index    Semantic   Name
--------  ---------- ------
       6  Classic    __default_bpdu_filter__
       8  Classic    protect-re
   17000  Classic    __default_arp_policer__
   57008  Classic    __cfm_filter_shared_lc__
[...]

NPC11(R2 vty)# show filter index 17000 program
Filter index = 17000
Optimization flag: 0x0
Filter notify host id = 0
Filter properties: None
Filter state = CONSISTENT
term default
term priority 0

    then
        accept
        policer template __default_arp_policer__
        policer __default_arp_policer__
                app_type 0
                bandwidth-limit 150000 bits/sec
                burst-size-limit 15000 bytes
                discard
```

Great! Now you know that the value of this default policer is 150Kbits/s max with a burst size of 15K bytes. Remember that this is a per-PFE policer, so it applies to all the ports attached to the PFE as an aggregate. To check the counters associated to this policer you can either call the global CLI command that will give you the sum of all PFE ARP policer instances (on all PFEs of the chassis) or call the PFE command that will give you the MPC point of view (the sum of only the ARP policer instance of the MPC). In both cases, the counter shows you the packets dropped and not the transmitted ones. In normal conditions the counters should be equal to 0 or stable.

The CLI command that gives you the global "policed" ARP packets is:

```
user@R2> show policer __default_arp_policer__
Policers:
Name                                            Bytes              Packets
__default_arp_policer__                         0                   0
```

And the same result, but at the PFE level for a given MPC:

```
NPC11(R2 vty)# show filter index 17000 counters <<< 17000 is the filter Index
Filter Counters/Policers:
   Index            Packets                 Bytes  Name
--------  -----------------  --------------------  --------
17000                   0              0  __default_arp_policer__(out of spec)
17000                   0              0  __default_arp_policer__(offered)
17000                   0              0  __default_arp_policer__(transmitted)
```

As mentioned, the counters only give you packet drops, which is why you don't see the 1pps ARP request flow. To illustrate the default policer in action let's replace the R1 router with a traffic generator and send a high rate of ARP requests. That should show you how the default ARP policer works:

```
user@R2> show policer __default_arp_policer__
Policers:
Name                                       Bytes         Packets
__default_arp_policer__                 141333400          286100

NPC11(R2 vty)# show filter index 17000 counters
Filter Counters/Policers:
   Index         Packets                 Bytes  Name
--------  --------------  --------------------  --------
   17000          286100     141333400  __default_arp_policer__(out of spec)
   17000               0             0  __default_arp_policer__(offered)
   17000               0             0  __default_arp_policer__(transmitted)
```

As you can see, the PFE and CLI commands give you the same result. Indeed, the ARP storm is not distributed over several MPCs, a good explanation of why PFE and CLI results are the same.

## A Custom per-IFL ARP Policer

With specific configuration you can override the default ARP policer on per-IFL (Interface Logical) basis, simply by configuring a policer and applying it on a specific IFL. This is the second of the three types of policers. And in this case a 100Mb/s ARP policer is configured on the ae0.0 interface:

```
user@R2> show configuration firewall policer my_arp_policer
if-exceeding {
    bandwidth-limit 100m;
    burst-size-limit 15k;
}
```

```
then discard;

user@R2> show configuration interfaces ae0
unit 0 {
    family inet {
        policer {
            arp my_arp_policer;
        }
        address 172.16.20.1/30;
    }
}
```

Now let's call back the following command to check which policers are applied on LAG's interfaces:

```
user@R2> show interfaces policers ae[0,1].0
Interface       Admin Link Proto Input Policer         Output Policer
ae0.0           up    up
                           inet
                           multiservice my_arp_policer-ae0.0-inet-arp
Interface       Admin Link Proto Input Policer         Output Policer
ae1.0           up    up
                           inet
                           multiservice __default_arp_policer__
```

As you can see, the ARP packets received on the ae1.0 interface still pass through the default ARP policer, while the ARP packets received on the ae0.0 interface are now rate-limited by the specific instance of the reconfigured policer. Let's replace R1 with a traffic generator again and send, one more time, the ARP storm on the ae0 interface. Recall the CLI command that shows you ARP drops:

```
user@R2> show policer | match arp
__default_arp_policer__                          0                    0
my_arp_policer-ae0.0-inet-arp             6099232750             12346625
```

As expected, the default ARP policer doesn't work anymore on ae0.0: it is the specific ARP policer that takes place.

## LU DDoS Protection ARP Policer

The third type of ARP policer is managed by the DDOS protection feature that provides hierarchical policing. First of all, you can check where the DDOS ARP policer is applied and its default value. To do that, use the following PFE command:

```
user@R2> request pfe execute target fpc11 command "show ddos policer arp configuration" | trim 5
DDOS Policer Configuration:

                                          UKERN-Config   PFE-Config

 idx prot        group        proto on Pri  rate burst   rate burst
 ---  --- ------------ ------------ -- -- ------ ----- ------ -----
 111 3400          arp    aggregate  Y Lo  20000 20000  20000 20000
```

Notice that the ARP packets (protocol_ID 3400) are policed as an aggregate, in other words, independently of the kind of ARP messages, at three levels: the LU chip, the µKernel, and the RE. In other words, at each level the ARP traffic will be rate-limited to 20kpps.

With this default configuration, the DDOS policer instance at the LU chip or the µKernel level should never drop ARP packets. Indeed, the default per-PFE ARP policer, seen previously, should rate-limit the ARP flow since it is more aggressive.

NOTE 150Kbits/s, which is the default PFE ARP policer, is equal to around 400pps (for standard ARP request messages).

With the default per-PFE ARP policer, you could experience some ARP DDOS protection drops but only at the RE level and in very rare conditions: for example, an ARP storm on every PFE on the chassis. Nevertheless, if you override the default per-PFE ARP policer with a less aggressive one, or if you disable it, DDOS protection could help you to protect the control plane against a massive ARP storm. Or, better, if you lower the DDOS protection ARP policer and make it more aggressive than the default per-PFE policer. The latter is actually a good practice, because as you are about to see the default 20kpps DDOS ARP policer settings are not very effective at the lower (PFE and microkernel) DDOS Protection levels.

CAUTION ARP entries need to be refreshed every 20 minutes. So if the PFE ARP policer or the DDOS ARP policer are set too aggressively, then the ARP replies needed for the ARP re-fresh may get dropped as well. Check the last paragraph of this Appendix for more information on how to address this (in a more sophisticated and scalable manner). Also, in big ethernet domains it is advisable to increase the ARP timeout value.

## Aggregate Microkernel Exception Traffic Throttling

For the sake of demonstration only, let's leave the default DDOS ARP configuration, and configure a custom per-IFL ARP policer at 100Mbit/s that overrides the default per-PFE ARP policer on ae0.0. Then, send 1 Gbit/s ARP storm on ae0.0.

As you've seen previously, the default per-PFE ARP policer on ae0.0 is no longer used, but the new 100Mbits/s per-IFL ARP policer rate limits the storm to 100Mbits/s.

Let's have a look at the DDOS violation notification:

```
user@R2> show ddos-protection protocols violations
Packet types: 198, Currently violated: 1

Protocol    Packet      Bandwidth  Arrival   Peak      Policer bandwidth
group       type        (pps)      rate(pps) rate(pps) violation detected at
arp         aggregate   20000      23396     1972248890 2014-06-17 10:18:12 CEST
          Detected on: FPC-11
```

As expected, 100Mbits/s is too much – representing around 23Kpps. The DDOS protection is triggered at the LU chip level. This first level of protection is rate-limiting the ARP to 20Kpps.

The LU chip delivers 20Kpps of not-to-be-discarded ARP request to the MQ/XM chip. These ARP requests are now conveyed within the Host stream 1151. Remember, the LU also assigns a Host hardware queue number. Let's check which HW queue is assigned for the ARP traffic (DDOS Proto ID 3400):

```
user@R2> request pfe execute target fpc11 command "sh ddos asic punt-proto-
maps" | match "arp" | trim 5

         contrl ARP                  arp aggregate    3400  2  20000  20000
```

The ARP traffic is conveyed in the Hardware queue 2. The base queue number of the Host stream on the MQ Chip is 1016 ( = Queue 0):

```
NPC11(R2 vty)# show cos halp queue-resource-map 0 <<<< 0 means PFE_ID = 0
Platform type      : 3 (3)
FPC ID             : 0x997 (0x997)
```

```
Resource init      : 1
cChip type         : 1
Rich Q Chip present: 0
  Special stream count: 5
  -------------------------------------------------

  stream-id  L1-node  L2-node  base-Q-Node
  -------------------------------------------------
       1151      127      254         1016
```

Let's have a look at the statistics of the Host Hardware Queue 1018 (1016 + 2), that the ARP traffic is assigned to, and check how many ARP packets are really sent to the µKernel. The ARP storm is being received on AE0, so you can use the MPC 16x10GE PFE-related command to retrieve this information:

```
NPC11(R2 vty)# show mqchip 0 dstat stats 0 1018 <<<< second 0 means QSys 0

 QSYS 0 QUEUE 1018 colormap 2 stats index 24:

                  Counter    Packets   Pkt Rate          Bytes   Byte Rate
 ------------------------ ---------- ---------- -------------- -----------
   Forwarded (NoRule)              0          0              0           0
   Forwarded (Rule)        188881107       9892   100667786767     5272436
   Color 0 Dropped (WRED)    6131890       1294     3268297370      689702
   Color 0 Dropped (TAIL)   40805679       8816    21749426459     4698928
[…]
```

Interesting! As you can see, a significant fraction of the 20kpps stream didn't reach the µKernel. The MQ/XM TOE conveys back presure in order to avoid µKernel congestion, and these drops are counted as Hardware input drops, as shown here:

```
user@R2> show pfe statistics traffic
[...]
Packet Forwarding Engine local traffic statistics:
    Local packets input                :      72044
    Local packets output               :      71330
    Software input control plane drops :          0
    Software input high drops          :          0
    Software input medium drops        :          0
    Software input low drops           :          0
    Software output drops              :          0
    Hardware input drops               :      36944 <<< sum of drops of the 8 queues of stream 1151
```

This is a proof that the default DDOS ARP policer has a too high value, and it is a good practice to make it more aggressive.

NOTE Early Junos implementation applies a 10kpps aggregate rate limit at the microkernel. This behavior is evolving from a hardcoded value to a dynamic assignment based on the microkernel's load.

The surviving 10kpps are queued in the medium software queue of the µKernel before their delivery to the RE. Call the following command to confirm this:

```
NPC11(R2 vty)# show ttp statistics
[...]
TTP Receive Statistics:
                   Control        High      Medium         Low     Discard
                ----------  ----------  ----------  ----------  ----------
 L2 Packets              0           0      100000           0           0
 L3 Packets              0           0           0           0           0
 Drops                   0           0           0           0           0
 Queue Drops             0           0           0           0           0
```

```
 Unknown                0          0          0          0          0
 Coalesce               0          0          0          0          0
 Coalesce Fail          0          0          0          0          0

[...]
```

And this confirmation is correct. You can use the tcpdump trace on the em0 interface again to confirm that this control packet is sent via the TTP protocol. Observe the DDOS Protocol_ID 0x3400 assigned by the LU chip and used by the jddosd process:

```
user@R2> monitor traffic interface em0 no-resolve layer2-headers matching "ether src host
02:00:00:00:00:1b" print-hex print-ascii size 1500
In 02:00:00:00:00:1b > 02:01:00:00:00:05, ethertype IPv4 (0x0800), length 122: (tos 0x0, ttl 255, id
40969, offset 0, flags [none], proto: unknown (84), length: 108) 128.0.0.27 > 128.0.0.1: TTP, type
L2-rx (1), ifl_input 325, pri medium (3), length 68
      proto unkwn (0), hint(s) [none] (0x00008010), queue 0
      ifd_mediatype Unspecfied (0), ifl_encaps unspecified (0), cookie-len 0, payload unknown
(0x00)
      -----payload packet-----
        unknown TTP payload
        0x0000: 0004 0004 3400 0000 ffff ffff ffff 0021
        0x000f: 59a2 efc2 0806 0001 0800 0604 0001 0021
        0x001f: 59a2 efc2 c0a8 0102 0000 0000 0000 c0a8
        0x002f: 0103 0000 0000 0000 0000 0000 0000 0000
        0x003f: 0000 0000
```

## RE DDoS Protection ARP Policer

When the ARP traffic reaches the RE, it is also rate-limited by the RE's ARP DDOS policer instance (the jddosd process). In this case, it did no trigger drops because the RE receives "only" 10Kpps (remember the default configuration of the ARP DDOS policer is equal to 20Kpps):

```
user@R2> show ddos-protection protocols arp statistics | find routing
Routing Engine information:
     Aggregate policer is never violated
     Received:  382746792         Arrival rate:    9894 pps
     Dropped:   0                 Max arrival rate: 9914 pps
       Dropped by individual policers: 0
 [...]
```

After passing through the DDOS policer, the ARP requests are "enqueued" by the system of the RE. Then the system schedules ARP "dequeue" to perform ARP request analysis, and finally generate the ARP reply if needed. Sometimes, and this the case in our storm's scenario, the arrival rate in the ARP system queue is too high and the system itself drops ARP requests. You can check system queue drop (interrupt drops) by calling this CLI command:

```
user@R2> show system queues | match "input|arp"
input interface     bytes        max  packets     max      drops
arpintrq              0       3000        0      50   144480486 < drops
```

NOTE Usually, you shouldn't observe any ARP interrupt drops if you keep the default per-PFE ARP policer, or set a low DDoS Protection rate for ARP – the example was just to show you that the Juniper MX Series router has several levels of protection.

System's drops and other ARP statistics can be collected by this other CLI command:

```
show system statistics arp
```

Finally, you can see ae0 interface statistics to check how many ARP replies are sent by R2:

```
user@R2> show interfaces ae0 | match rate
  Input rate     : 954910752 bps (234968 pps)
  Output rate    : 1667520 bps (4342 pps)  <<<< ARP replies
```

In conclusion, R2 received an ARP storm of 230 Kpps (1Gbits/s) but your custom per-IFL ARP policer of 100Mbits/s has reduced the attack to 23Kpps. The DDOS policer instance of the LU chip has rate-limited the attack to 20Kpps. Then only 10kpps are really delivered from the MQ/XM chip to the µKernel due to the back pressure coming from the TOE. Finally, the RE operating system rate-limits itself the 10Kpps ARP. It handles "only" around 4.2Kpps.

MORE? Default (per-PFE, DDOS, etc.) ARP policers are stateless and can drop some valid ARP packets. Also, custom policers are cumbersome to use and if set too aggressively they can drop legitimate ARP replies. One solution is to turn on Suspicious Control Flow Detection (SCFD). The SCFD feature allows control traffic to be rate-limited in a per-interface basis. It is a powerful extension of the DDoS protection feature, and it is not covered in this book. Check out Juniper's tech docs at http://www.juniper.net/documentation.