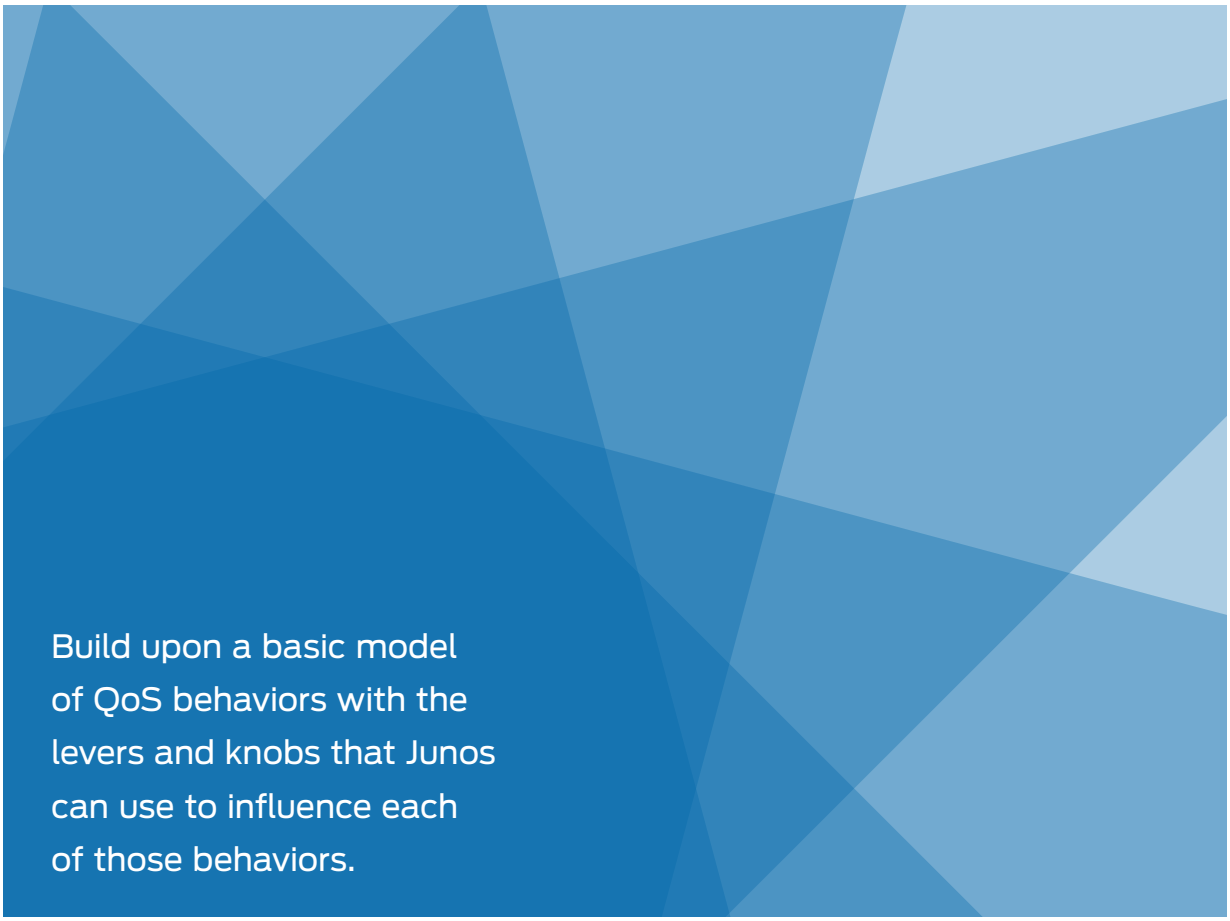
JUNIPER NETWORKS

Junos® Fundamentals Series

DAY ONE: DEPLOYING BASIC QoS



By Guy Davies

DAY ONE: DEPLOYING BASIC QoS

The demands being placed upon today's networks are growing at an incredible rate. Given the rapid increase in the number of attached devices, the explosion in traffic generated by these devices, and the convergence of legacy networks designed to carry a single type of traffic in isolation – the old approach of simply overprovisioning to support the potential peaks of data is no longer commercially or technically feasible.

To stop this perfect storm of a log jam, *Day One: Deploying Basic QoS* gives you an overview of Quality of Service (QoS) concepts and then provides tools and techniques from the Junos operating system toolbox to implement a comparatively simple class-of-service configuration. It's a start, it works, and it can be done in your test bed on day one. And true to the principles of *Day One* network instruction, you'll be guided through a set of basic requirements and configuration tools using multiple templates and examples from which you can derive your own valid configurations.

"This book is a must have for anyone seeking to configure QOS in any Juniper device due to its clarity, precision, and ease of use. It's applicable to a wide range of engineers, from the Junos novice all the way to the expert. Guy can't help but share his immense knowledge and practical experience, adding extra value to the topic and the book as a whole."

Miguel Barreiros, Senior Professional Services Consultant, Juniper Networks

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

- Understand the principles of QoS, independent of any vendor's implementation.
- Identify the basic building blocks of a QoS implementation.
- Identify common traffic behaviors and how they can be manipulated.
- Construct combinations of the basic building blocks in order to induce a required behavior.

Juniper Networks Books are singularly focused on network productivity and efficiency. Peruse the complete library at www.juniper.net/books.

Published by Juniper Networks Books

ISBN 978-193677930-7

51600

9 781936 779307

07500213

JUNIPER NETWORKS

Junos® Fundamentals Series

Day One: Deploying Basic QoS

By Guy Davies

JUNIPER NETWORKS

Published by Juniper Networks Books
Author: Guy Davies
Editor in Chief: Patrick Ames
Copyeditor: Nancy Koerbel
J-Net Community Management: Julie Wilder

ISBN: 978-1-936779-30-7 (print)
Printed in the USA by Vervante Corporation.
ISBN: 978-1-936779-31-4 (ebook)

Version History: v2 July 2015
3 4 5 6 7 8 9 10 #7500213

About the Author
Guy Davies is a Senior Solutions Consultant in the Global PS Mobile Core Networks organization at Juniper Networks. He has worked for Juniper Networks for five years in the EMEA and Global PS organizations, helping Service Providers and large enterprises to build networks delivering large scale, high availability, and granular quality of service to their customers. Prior to Juniper Networks, Guy spent six years working for Telindus, a Systems Integrator, and prior to that five years at UUNET in the UK. In these roles Guy has delivered large scale MPLS and IP core and provider edge networks, subscriber management platforms, and AAA platforms, all of which have placed an ever increasing emphasis on differentiated Quality of Service. Guy is JNCIE-M #20.

Author's Acknowledgments
I would like to thank my family for their apparently inexhaustable patience. I would also like to thank the Juniper Networks Books team for their support and drive to get this book written and knocked into shape. Without them, it wouldn't have been completed. Finally, I would like to thank my colleague, Miguel Barreiros, for his advice and review of this book.

This book is available in a variety of formats at: www.juniper.net/dayone, as well as on iTunes and Amazon.

Send your suggestions, comments, and critiques by email to dayone@juniper.net.

What You Need to Know Before Reading This Book

- You should have a solid understanding of the principles of packet-based networking.
- It is beneficial if you are familiar with the Junos Command Line Interface. It is also useful to read other *Day One* books in the *Junos Fundamentals Series*.

After Reading This Book, You'll be Able To

- Understand the principles of QoS, independent of any vendor's implementation.
- Identify the basic building blocks of a QoS implementation.
- Identify common traffic behaviors and how they can be manipulated.
- Construct combinations of the basic building blocks in order to induce a required behavior.

Why QoS?

The demands being placed upon networks today are growing at an incredible rate. With the rapid increase in the number of attached devices, the explosion in traffic generated by each of those devices (particularly from video applications) and the convergence of multiple legacy networks designed to carry a single type of traffic in isolation, the old approach of simply overprovisioning to support the potential peaks of data is no longer commercially or technically viable. Subscribers of certain services (e.g. telephone services) demand that those services are always available and also of an acceptable quality. In order to ensure that availability and quality, it is first necessary to group traffic into classes where traffic in a single class requires the same treatment, and then to ensure that treatment is delivered consistently to all traffic in that group. This consistency is required not just in a single device but in all devices that the traffic crosses from source to destination.

This book aims to give the reader an overview of the terminology of QoS and then to provide some tools and techniques from the Junos operating system to allow the reader to implement a comparatively simple class-of-service configuration. The title of this book is *Deploying Basic QoS*, and it is certainly not an attempt to provide a complete insight into every class-of-service option on every single platform sold by Juniper Net-

works. For that, the reader can refer to the documentation available at http://www.juniper.net/techpubs/software/junos.

This book is intended to guide the reader through the basic requirements and configuration tools, using templates and examples from which they can derive their own valid configurations. There are plenty of aspects of class-of-service configuration that are completely absent from this book. These are left for more advanced publications.

MORE? A fabulous resource for QoS is the newly published *QoS Enabled Networks: Tools and Foundations*, by Peter Lundqvist and Miguel Barreiros, (John Wiley & Sons, 2011, ISBN 978-0-470-68697-3), two senior engineers at Juniper Networks. For more information about the book and its contents, visit your favorite online bookseller, or www.juniper.net/books.

Chapter 1

Introducing QoS

This first chapter examines the fundamental principles of Quality of Service (QoS), starting with the basic idea of the end-to-end user experience and graduating to the way in which QoS is implemented as a series of hop-by-hop behaviors. The chapter builds a basic model of QoS behaviors, including those which have been standardized, and describes the "levers" that can be used to influence each of those behaviors.

Quality of Service Versus Class of Service

There are many possible definitions of QoS, but for the purposes of this book, Quality of Service (QoS) is the manipulation of *aggregates of traffic* such that each is forwarded in a fashion that is consistent with the *required behaviors of the applications generating* that traffic.

From an individual user's point of view, QoS is experienced on the end-to-end (usually round trip) flow of traffic. However, it is implemented as a set of behaviors at each hop – this is an important distinction that is absolutely fundamental to QoS, and it is critical that the reader understands it clearly.

In effect, this means that a single hop with no configured QoS can destroy the end-to-end experience and nothing that subsequent nodes do can recover the end-to-end quality of experience for the user. That doesn't mean that QoS *must* be configured at every hop. However, it's critical to understand that a single congested hop can be the undoing of the most intricate QoS design.

On the other hand, Class-of-Service (CoS) is a configuration construct used within the Junos operating system to configure an individual node to implement certain behaviors at that node, such that the end-to-end QoS is consistent with the desired end-to-end user experience or application behavior.

Each class is associated with an aggregate of traffic that requires the same behaviors as it flows through the network device. Classes do not relate implicitly to traffic belonging to a single application; rather, any application requiring the same behaviors generates traffic belonging to the same class.

TIP Does the difference between QoS and CoS make sense? If not, reread these few introductory paragraphs again. The concept is the foundation for the entire book and is often obfuscated in QoS literature.

What are Behaviors?

You have already read about behaviors and you just started Chapter 1. That's because they are a core concept in QoS. While the definition of behaviors should be familiar, the concept, in terms of QoS, may not. Let's take a little time to explore exactly what is a QoS behavior.

A QoS behavior *describes the way in which a particular flow of traffic expects to be handled* as it passes through each network device. This is usually expressed in terms of three characteristics that are particularly relevant to certain *classes of traffic*. The three characteristics are:

- *Loss*: This is the failure of a packet, which was transmitted into the network at its source, to reach its intended destination.
- *Latency*: This is the delay between the transmission of a packet into the network at its source and its arrival at its intended destination.
- *Jitter*: This is the variation in latency between consecutive packets in a single flow.

These three characteristics are generally used to describe the quality of service associated with traffic belonging to a particular application travelling end-to-end. They are also the characteristics that you can manipulate (sometimes indirectly) on a hop-by-hop basis in order to create the per-hop behaviors you want, and to ensure the traffic receives the desired end-to-end QoS.

Needless to say, each of the three characteristics can have a significant impact on particular applications. Let's investigate each one.

Loss

This is the failure of a packet, which was transmitted into the network at its source, to reach its intended destination. Loss can be induced by many factors including errors, link and node failures, and congestion in the network, or, indeed, by an intentional action on any of the nodes in the network. While it is important to understand the actual cause of the loss in order to be able to effectively manipulate it, in terms of the perceived QoS, the cause of loss is generally unimportant.

That's because when a packet is lost, there are two possible consequences: either the loss can be ignored by the application (maybe the application is able to deduce the information in the lost packet, or the

application is tolerant of a single loss), or the packet must be transmitted again. If the packet must be transmitted again, either the transport layer provides a mechanism for reliable transmission (for example, TCP) or it is the responsibility of the application to request a re-transmission.

An example of an application that may be tolerant of loss is audio, as shown in Figure 1.1. A lost packet in an audio stream may result in a very short silence, or an audible *pop*, and in this sense, the application "fails" as shown in the lower line of symbols; but the human ear and brain are able to compensate for these small gaps or distortions, so it is unnecessary to compensate within the network for low-level packet loss in an audio application.

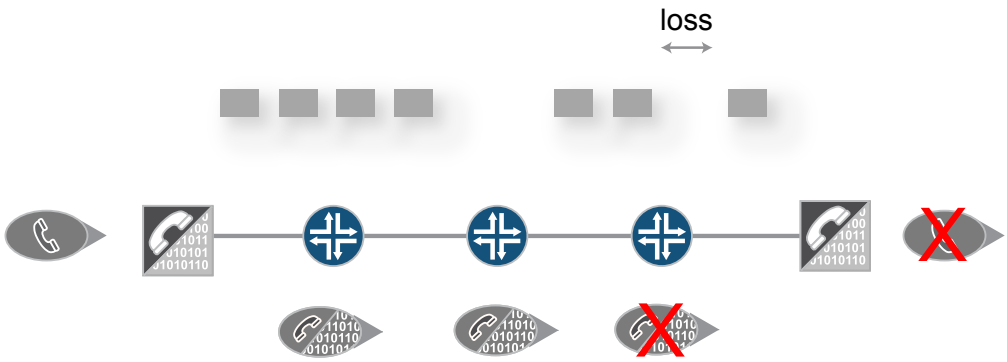


Figure 1.1 Simplified Representation of the Impact of Loss on a Digitized Analogue Signal

Conversely, the banking system and its network is incredibly intolerant of loss and is an example of an application that can not tolerate loss. Imagine if the data regarding the transfer for your monthly salary is in the packet that is lost, and you lose a zero at the end. Your 1000 becomes a miserly 100. This cannot simply be ignored; it must be identified and the packet must be retransmitted.

Latency

Latency is the *delay between transmission and receipt of a packet*. As shown in Figure 1.2, latency in many applications is of little consequence.

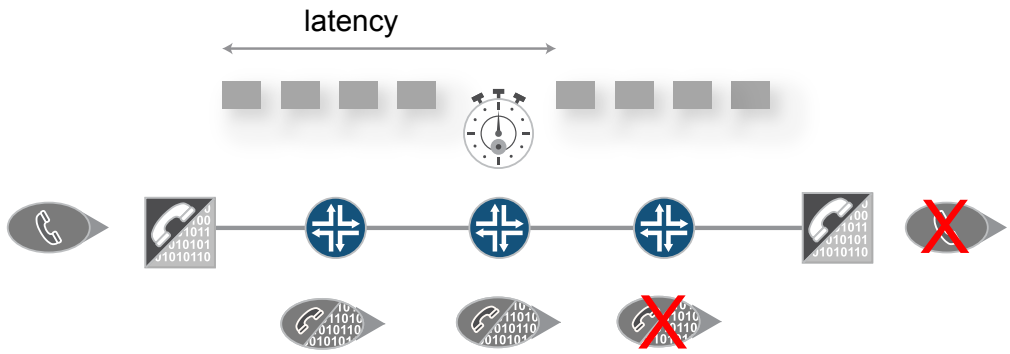


Figure 1.2 Simplified Representation of Causes and Impact of Latency

For example, the transmission of Internet radio is so heavily impacted by encoding delays that the latency introduced by the network is not likely to be considered a significant additional problem. The time-checks given out by the DJ are delayed anywhere between 1 and 15 seconds anyway, and are so variable as to be of somewhat limited value.

You know how disruptive it is when latency is experienced on a voice call, however, because it's necessary to wait for a quiet period in order to start speaking. The likelihood of two people talking at the same time always seems quite high. The human ear and brain can tolerate around 200ms of latency with no noticeable trouble. At above 500ms, the delay becomes noticeable enough to be a problem, which makes the call more difficult.

NOTE Both of these examples are audio streams, but the unidirectional nature of Internet radio makes it much more tolerant of latency, whereas with an interactive application the latency chips away at the interactivity until it can almost be static.

Jitter

Jitter is the *variation in the amount of latency in consecutive packets*, and it has the most significant impact on some of the most highly valued services, such as voice and video services. Voice services, in particular, rely upon the digitization of the analogue voice signal into chunks of data that can be transmitted in packets and then, at the far

end, reassembled into an analogue stream. Normally, that digitization process produces a steady stream of packets with a constant time between each packet. At the receiving end, there is a buffer of fixed length into which packets are placed until enough packets are present to decode the next section of analogue signal. If, during transmission, the latency of consecutive packets varies such that the time between the arrival of consecutive packets differs too much, then the conversion back to the analogue signal fails because the required packets are not present in the buffer at the time required for them to be converted into a meaningful analogue signal. Consider Figure 1.3 as a visual example.

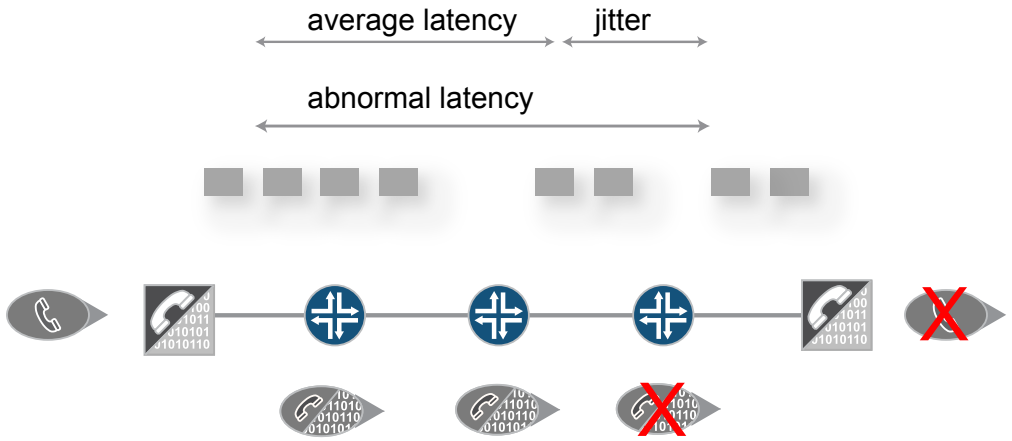


Figure 1.3 Simplified Representation of the Impact of Jitter on a Digitized Analogue Signal

You can see in Figure 1.3 that the impact of jitter can be reduced by extending the de-jitter buffer. It is assumed that the jitter will be less than the maximum length of the de-jitter buffer. The downside of this approach is that it implicitly adds latency, which, as already discussed, is also bad for interactive voice applications.

Summary

That's it. Three little behaviors that are the sum of QoS around the world and around the world's networks. Come back to this chapter and its simplified definitions when, or if, you get confused as you try to adjust individual nodes on your network to fine-tune for the many applications and the three traffic behaviors they may exhibit.

Chapter 2

Basic Junos QoS Concepts and Packet Flow Through Routing Nodes

Chapter 1 explained the basic concepts of QoS, CoS, and Behaviors. Now Chapter 2 examines the basic building blocks of a Junos CoS configuration, and then shows the packet flow through the various QoS functions, which are (almost) universal in Junos routing, switching, and security platforms. It then maps those QoS functions onto the packet flow through some of the different Junos hardware platforms, focusing on a few current platforms.

TIP Remember that each network device behaves more or less independently of all other network devices, so the only things you can actively influence are *per-hop behaviors*.

The Building Blocks of a Junos CoS Configuration

In subsequent sections, this book focuses on Junos CoS as implemented on the M/T Series Routing Nodes, the MX Series Ethernet Services Routers, and the SRX Security Nodes. These are simply used as current examples of core, edge, and security nodes.

In every Junos CoS implementation there are certain functions that are required in order to be able to influence the behavior of outbound packets on a particular interface.

NOTE Each vendor's networking equipment implements the control of these functions in different ways, and may use slightly different terminology. The terminology used in this book, and defined in this chapter, is the terminology used in Junos configurations, but the explanations should be sufficiently vendor-agnostic as to be broadly applicable to different vendors' equipment.

Let's first list our key Junos CoS functions that can influence the behavior of outbound packets, and then devote a short section to each:

- Forwarding Class
- Classification(s)
- Policing
- Random Early Discard (RED)
- Shaping
- Scheduling
- Remarking

Forwarding Class

A forwarding class is a label, used entirely within a network node, which is used to identify all traffic that requires a single behavior when leaving that node. Forwarding classes do not explicitly appear outside a node, although if the QoS configuration of all nodes in a network is consistent, it can easily be derived from information in packet headers.

Classification

Classification is the act of identifying the class to which a packet belongs. It is usually initially performed on ingress to each node, although a packet may be reclassified at various points on its path through a network node.

In Junos there are three main approaches to classifying packets, which vary in their degree of flexibility and in the complexity of the required configuration: Interface Based Classification, Behavior Aggregate (BA) Classification, and Multifield (MF) Classification. These approaches are not all mutually exclusive, and, in some combinations, can be applied in series to get a less granular first-pass behavior, followed by a more granular reclassification of a subset of the traffic.

Interface Based Classification

If all traffic arriving on a single interface is known to be associated with a single class then the easiest mechanism to classify this traffic is simply to associate all traffic arriving on the interface with the relevant forwarding-class.

While somewhat trivial to implement, this method assumes that all traffic arriving on the interface is of the same class. There is no inherent mechanism to indicate any exceptions, so it is very inflexible. It can be used in conjunction with MF classifiers, however, to provide more granular exceptions to the default interface classification if required.

TIP This mechanism is also useful if the upstream node is *untrusted* and you wish to *bleach* all traffic coming in by applying a single class (usually Best Effort in this situation).

Behavior Aggregate Classification

Behavior aggregate classification (BA) provides a good balance between flexibility and complexity. It is particularly attractive where the traffic being classified is being transported in large aggregates (for example, in the core of a network, where traffic associated with many unique applications passes over a single link, making Multi-Field classification unattractive). BA Classification relies upon markings placed in the headers of incoming packets: either Ethernet frames, IPv4 or IPv6 packets, or MPLS frames. Each of these packet or frame types includes a field in the header specifically designated for the indication of a class to which this packet has been previously assigned.

In Ethernet (using 802.1Q VLAN frames) there are three 802.1p bits. In IPv4 packets, there is the Type of Service Byte from which you can either use the three precedence bits, or six bits to indicate the DiffServe Code Point (DSCP). IPv6 has six bits of the IPv6 DSCP and MPLS has the three *experimental* bits.

NOTE There are actually 8 bits in IPv6, but two have been reserved for future use.

NOTE To use the term *experimental* bits for MPLS is something of a misnomer, since this utilization of these three bits is no longer experimental in any sense. No other use has been proposed for these three bits, and there are efforts in place to rename them to something more appropriate to their current function.

It's important to note that the main constraint with this model is that the upstream node must be *trusted* to correctly (and fairly) mark packets. If the upstream node cannot be trusted then it could be a concern that the node would mismark traffic into a class that would receive a higher QoS than it requires, or for which the owner of the upstream node has paid.

Multifield (MF) Classification

The most flexible, but also the most complex, classification to configure and maintain is the Multifield (MF). It uses firewall filters (also known as *access-lists*) to identify arbitrary attributes of an IP packet (it is less commonly applicable to non-IP traffic types) and places traffic into a particular traffic class based on the contents of the IP packet.

Since this approach is effectively only constrained by the characteristics that can be matched in a firewall filter, it is possible to be very granular in the choice of traffic class to which the packet belongs. However, granular choices require comparatively complex filters, which may have to be customer specific. This degree of complexity and administrative overhead makes the use of MF classifiers particularly attractive where the upstream node is not trusted (or not able) to mark the packets, and the requirement to apply QoS based on arbitrary parameters is strong.

In addition, MF classifiers can be used to modify the forwarding-class selected by a BA classifier or an interface classifier. Thus, as mentioned before, it is possible to make a *rough* classification based on the BA markings (or on an interface marking) and then reclassify a subset of the traffic based on arbitrary attributes in the IP headers.

Policing

Policing is the method of applying a hard limit to the rate at which traffic can access a resource (for example, upon entry to a node or to a queue on egress). Since a policer constrains access to the node or queue, once a decision is made that a packet is non-conforming and that it should not gain access to the protected resource, that packet will be dropped (or reclassified). This hard-drop behavior can have a negative impact, particularly on TCP traffic, and particularly when the policer is run consistently at its limit.

While it is possible to reclassify packets based on a policer, it is important to be very careful to avoid reordering of packets in applications that may be sensitive to the order in which packets are received.

In Junos, policing can operate in three modes:

- A simple policer operates based on a single rate-limit and a single burst-size. This is also known as a single-rate, two-color policer.
- A single-rate, three-color policer uses a single rate-limit but has two burst sizes. This provides a mechanism to create three loss-priorities (as described for Assured Forwarding in RFC2597).
- Two-rate, three-color policers use two rates, a committed rate and a peak rate, to achieve the same results as a single-rate, three-color policer.

Random Early Discard

Random Early Discard (RED), also known as *Random Early Detection*, is a congestion avoidance mechanism. It helps to mitigate the impact of congestion (specifically with TCP-based traffic).

MORE? For a really thorough review of the behavior of TCP in the presence of congestion, and why RED can help avoid some of the worst aspects of that behavior, see *QoS Enabled Networks: Tools and Foundations*, by Peter Lundqvist and Miguel Barreiros, (2011, John Wiley & Sons), at your favorite online bookseller or via www.juniper.net/books.

By selecting random TCP packets from a queue and discarding them, the end point that was awaiting delivery of that packet fails to send an acknowledgement (or, if that packet was an acknowledgement, the far end does not receive the acknowledgement) for the packet. This triggers retransmission of the packet and the reduction of the transmission window size (and as a consequence the speed with which the source transmits TCP packets). The random nature of the selection of the packets to be dropped ensures that no single flow of traffic, application, or source is unfairly penalized and every source continues to get its "fair share" of the available capacity on a link that is close to congestion.

Since the TCP source from which the packet was dropped slows down the rate at which it transmits packets, the degree of congestion is reduced.

Thus, you have a mechanism that is "fair" to all. But QoS is not necessarily about being fair to all, it's about ensuring that high priority (*high value*, *loss-*, *latency-*, or *jitter-sensitive*) traffic is given priority. In order to manipulate the rate at which packets belonging to particular forwarding-classes are dropped, it is necessary to apply a weight to RED for each forwarding-class. This process is known as *Weighted RED* (WRED). It is particularly important to apply a weight to RED in order to avoid dropping packets in forwarding-classes that are particularly intolerant of loss (for example, expedited forwarding and assured forwarding).

NOTE Expedited forwarding and assured forwarding are defined behaviors, the definitions of which can be found in RFC3246 and RFC3260 respectively.

Often, the traffic associated with applications that are particularly intolerant of loss, latency, and jitter are transported in UDP. In this case, the application of RED is counterproductive since it damages perceived QoS of the application. In addition, since UDP has no built-in mechanism to identify the loss of a packet and modify its rate of transmission, the packet is either simply lost as a consequence, (reducing the perceived QoS) without having any significant impact on the throughput, or worse, the application identifies the loss and demands retransmission of the packet anyway, so the packet is then seen twice, potentially *increasing* the congestion.

Shaping

Shaping is the application of a limit to the rate at which traffic can be transmitted. Unlike policing, it acts on traffic that has already been granted access to a queue but which is awaiting access to transmission resources. Traffic that does not conform to the shaper's criteria is generally held in the queue until it does conform, and no explicit constraint is placed upon more traffic entering the queue (as long as the queue isn't entirely full). Therefore, shaping can be less aggressive than policing and can have fewer of the negative side effects.

A shaper is normally defined in terms of a Committed Information Rate (CIR) and/or a Peak Information Rate (PIR).

Scheduling

Scheduling is the act of deciding the order in which to place packets onto the wire based upon the class to which they belong (or the queue in which they're waiting). Given that you have multiple queues, all of which may contain packets waiting to be transmitted, but you only have a single serial transmission media, you have to decide which queue to service first, for how long, and with what frequency you return to check whether each queue has traffic to send.

Remarking

As mentioned above, Ethernet, MPLS, IPv4, and IPv6 packets all have a field in the header that can be used to inform another node about a classification decision made earlier in the path. Remarking is the act of (re)placing a value in the header of an outgoing packet, which identifies

the class to which the packet was assigned by the transmitting router. Subsequent nodes can use this marking to easily and consistently classify the packet using a BA classifier. It is possible to remark each of the packet header types using each of the marking types (IEEE 802.1p, MPLS EXP, IPv4 Precedence, IPv4 DSCP, or IPv6 DSCP) that can be used by BA Classifiers.

Packet Flow Through the CoS Functions

Figure 2º.1 shows the flow of the packet through the various CoS functions in Juniper Networks routing, switching, and security nodes. At the top are the functions performed on the ingress hardware moving to the right, while along the bottom are the functions performed on the egress hardware with the packet moving to the left. The box in the middle represents the storage of the forwarding-class and loss-priority, the two values that can be manipulated during the flow of the packet through the router, and based ultimately upon which treatment of the packet (in the last two boxes) is undertaken.

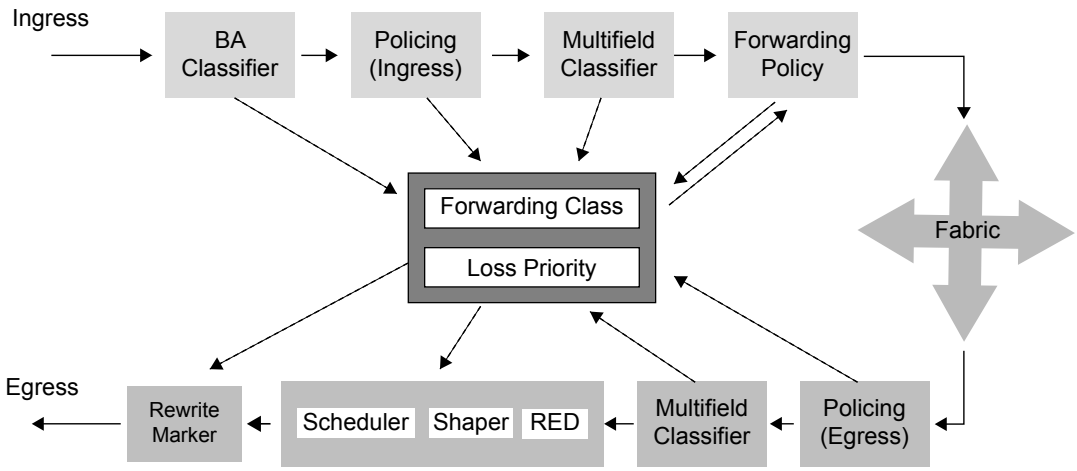


Figure 2.1 Junos CoS Processing

You should recognize the labels on almost all of the boxes from the descriptions given in this chapter and in Chapter 1. If not, quickly review their functionality.

NOTE The *BA Classifier* box in Figure 2.1 includes both the BA Classifier function as described and the interface classifier. It is not possible to apply both styles of BA classifier simultaneously to a single interface.

NOTE In Junos, the Policing (Ingress) and Multifield Classifier are implemented using the same firewall filter construct, so a single firewall filter can act first to police any non-conforming traffic then to apply a forwarding-class and loss-priority to any conforming traffic as defined in the firewall filter actions.

Before being transmitted onto the switch fabric of the network device, the traffic can be subjected to a *Forwarding Policy*. This is implemented as another firewall filter, which acts upon traffic as it is *about to enter* the switch fabric based upon information in the forwarding tables along with the existing forwarding-class and loss-priority. Note the bidirectional arrows center Figure 2.1 between the forwarding policy and the box in the center of the diagram.

On egress, it is again possible to manipulate the forwarding-class and loss-priority of the outgoing packet before it is queued. This is achieved using another Policing (Egress)/Multifield Classifier combination implemented as a firewall filter, exactly as on the ingress.

Once policed and classified for a final time, the traffic is queued where it is then acted upon by the RED profile, any Shaper, and then the Scheduler.

Finally, just before transmission onto the wire, any markings in the Ethernet, MPLS, IPv4, or IPv6 headers are modified by a Rewrite rule. This helps a downstream networking device to make a classification decision more easily, even if the packet is now part of a massively aggregated flow.

Packet Flow Through Hardware

The packet forwarding architecture of Juniper Networks routers has evolved significantly since the M40 was first released in 1998. However, the basic architecture of the routers has not changed. Every router is split into three elements, the *control plane* (this function is performed by one or more routing engines), the *forwarding plane* (this function is performed by one or more packet forwarding engines), and the services plane (this function is performed by one or more Services PICs or DPCs). The packet flow through these elements is shown in Figure 2.2.

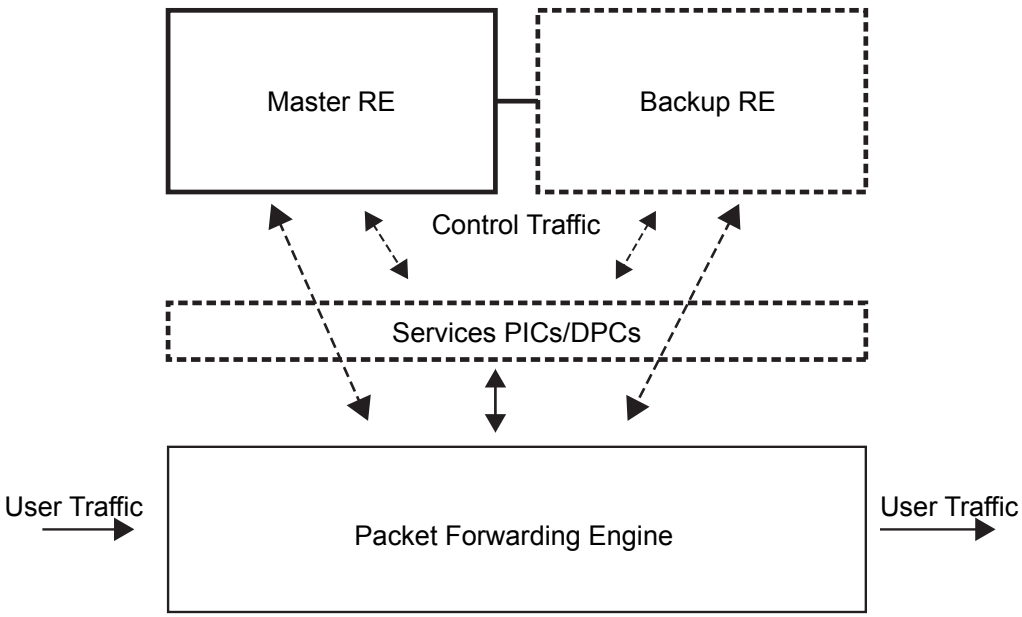


Figure 2.2 Packet Flow Through the Routing Node

The packet forwarding plane (PFE) is the element that has evolved the most – from a single, shared memory architecture on the M40 and the subsequently released M20, M5, M10, M7i, and M10i, to the M160 and M120, with the PFEs on the Switch Fabric Cards, to the M320, T series, and MX series where one or more complete PFE complexes were placed on the linecards and multiple switch fabrics provided highly resilient paths between the PFEs – the concept has remained the same: *user traffic must be forwarded independently of the load on the control plane (the RE)*.

With the development of each of the new PFE hardware, class-of-service has been enhanced, culminating in the current range of MX3D platforms based on the Junos Trio chipset (the platform that provides scaling in the number of services delivered at high capacity to a large number of subscribers). And it is that scaling of subscribers, services, and bandwidth that places increasing focus upon the QoS design.

At a very high level, the packet flow through the hardware is consistent between each of the platforms. The number of PFE complexes and PICs on a single linecard may differ, as may the number of switch fabrics, as shown in Figure 2.3, but the concept remains the same.

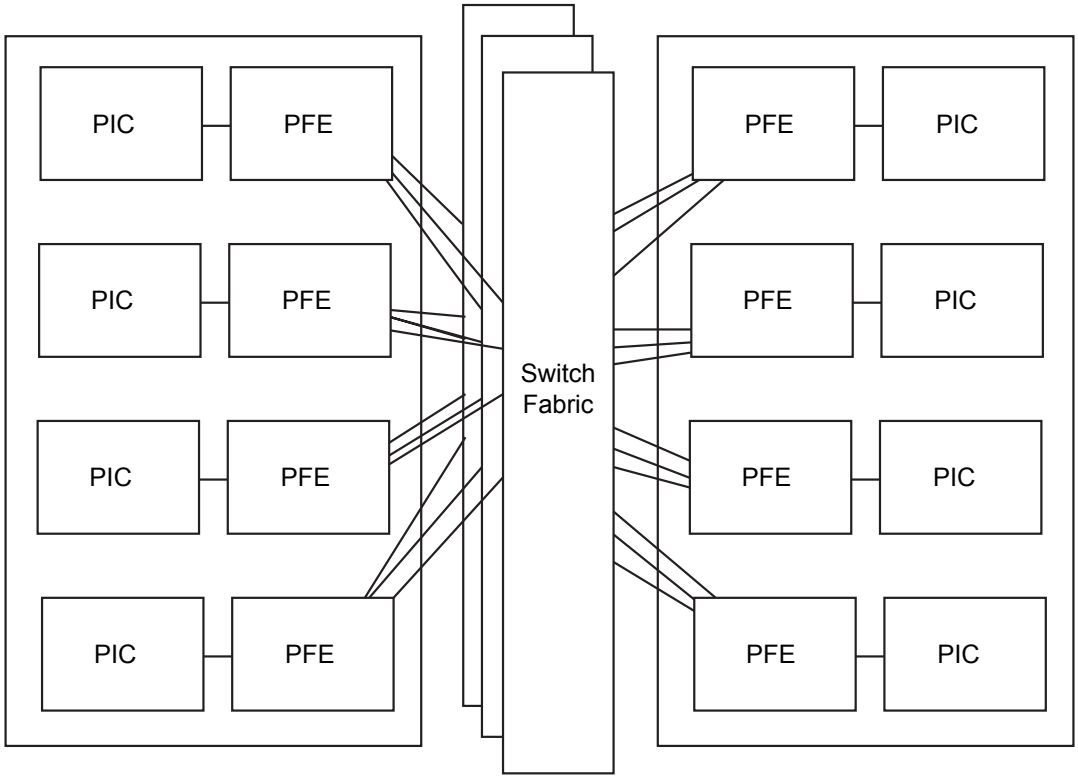


Figure 2.3 PFE Complexes and PICs on a Linecard

The packet flow through hardware follows this sequence:

- A packet arriving on a PIC is parsed (at Layer 2) and the packet, along with information relating to the Layer 2 information obtained by the PIC, is passed to the PFE.
- The ingress PFE then parses the remaining header information, creating a fixed length block of metadata describing the packet and, depending upon the router and linecard, may break the data portion of the packet into chunks for temporary storage.
- Within the PFE, the packet is classified with a forwarding-class and packet loss priority. In addition, a decision is made regarding the PFE to which the packet must be forwarded. This decision is based not only on the destination of the packet, but also on any firewall filters that may be matched by the packet, the forwarding-class, and the packet loss priority (PLP) of the packet.

- The ingress PFE then requests resources from the egress PFE in order to transmit the packet and metadata across the switch fabric. The packet may be sent over multiple switch fabrics.
- At the egress PFE, the packet is reassembled, ensuring that the entire packet is correctly assembled and that packets are returned to the order in which they were transmitted (in the case that they became re-ordered over the switch fabric).
- The egress PFE then performs exactly the same steps, parsing the packet – making a forwarding decision based on egress policers, drop-profiles, schedulers, and shapers – and finally rewriting the headers so that the packet can be transmitted onto the wire with the appropriate markings.

You might have noticed from Figure 2.3 that even if a packet arrives on a PIC in one linecard, and is destined for another PIC in the same linecard, that packet will cross the switch fabric. The only exception is the case where a packet arrives on one port on a PIC and is destined for another port on the same PIC, in which case, on some platforms, the packet takes the shortcut between the ingress functional blocks of the PFE and the egress functional blocks of the same PFE. In all cases, however, the packet must go up to the PFE in order to be switched between ports.

Summary

So now you should now have a clear understanding of the basic elements that allow you to manipulate QoS, the behaviors that these elements influence, the hardware functions on which they are implemented, and in which order they are implemented in Juniper Networks platforms.

Next, we move on to take a look at exactly how you configure each of these elements in order to build a relatively simple, but complete, CoS configuration in Junos.

Chapter 3

Building a Basic QoS Implementation Using Junos Software

This chapter describes how to take each of the basic QoS functions and combine them at various points in the packet flow in order to deliver a consistent and flexible QoS design.

Each section includes a template configuration that could be used to configure that function. The templates contain variables in the form:

```
$variable_name$
```

These variables must be replaced with valid values in order to complete the configuration. Some elements may appear multiple times in a valid configuration, but in the interests of brevity, *will appear only once in the template*.

As a point of reference, Figure 3.1 is the topology this book uses. Your network will, of course, be different, but if you can build a follow-along test bed mimicking this simple topology, you should be able to better follow along as a lab exercises.

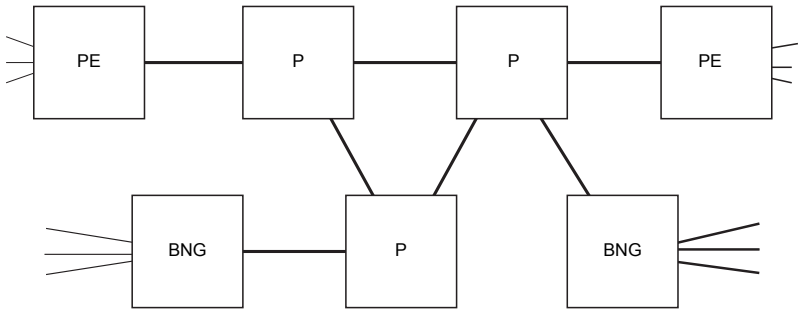


Figure 3.1 Topology Used for This Book

Code Points

In order to begin our QoS implementation, first it's necessary to identify which type of code points you will be using. It is highly likely that in your P routers, and on core facing interfaces on your PE routers, you will be using BA classification. Therefore, you must decide whether you need to specify aliases for your code points – aliases simply provide a human friendly name for a numeric value. These names may be easier to remember or more easily associated with particular forwarding-classes, but they are entirely optional. A default set of aliases is provided for each of the marking types.

In general, it isn't necessary to create your own aliases, but should you want to, the template for configuring aliases is as follows:

```
class-of-service {
   code-point-aliases {
      $marking_type$ {
         $alias$ $code_point_bits$;
      }
   }
}
```

Choosing a Classification Approach

For each ingress interface, it is necessary to choose between the three possible classification approaches:

- Interface Classfiers
- Multifield (MF) Classifiers
- Behavior Aggregate (BA) Classifiers

While Chapter 2 discussed the different compromises between complexity of administration and flexibility of function, the examples below demonstrate the use of each style of classifier in addition to pointing out where it is used and why that choice was made.

Configuration Template for Interface Classification

In the following template, you can see the `forwarding-class $class_name$` being applied to a logical interface `$interface_name$.$unit_id$`. You should note that this is all applied under the `[edit class-of-service]` hierarchy level:

```
class-of-service {
   forwarding-classes {
      class $class_name$ queue $queue_number$;
   }
   interfaces {
      $interface_name$ {
         unit $unit_id$ {
            forwarding-class $class_name$;
         }
      }
   }
}
```

Notice, too, that there is a template for the definition of `forwarding-classes` (the configuration options for forwarding-classes are discussed in more detail later in this chapter). The `forwarding-classes` template element is included here purely to indicate that it is linked to the `$class_name$` used in the `interface` configuration.

Configuration Template for Behavior Aggregate Classification

This template shows the three elements required to configure and apply a Behavior Aggregate classifier. The first element is the ubiquitous forwarding-classes definition:

```
class-of-service {
   forwarding-classes {
      class $class_name$ queue $queue_number$;
   }
   classifiers {
      $marking_type$ $classifier_name$ {
         class $class_name$ {
            loss-priority $loss_priority$ code-points [$code_points];
         }
      }
   }
   interfaces {
      $interface_name$ {
         unit $unit_id$ {
            classifiers {
               $marking_type$ $classifier_name$;
            }
         }
      }
   }
}
```

NOTE The forwarding-classes only have to be defined once for the entire configuration. They are repeated here simply to show that they are required for each of the three methods.

For the second element, each classifier is associated with a specific `$marking_type$` (`ieee-802.1`, `exp`, `inet-precedence`, `dscp`, or `dscp6`) and for each code-point, or set of code-points, a forwarding-class `$class_name$` and a loss-priority `$loss_priority$` are applied to the packet.

And the third element required is the application of the classifier to a logical interface `$interface_name$.$unit_id$`. The entire configuration is applied under the [`edit class-of-service`] hierarchy level.

Configuration Template for Multifield Classification

Multifield classifiers are somewhat implementationally different than the other two types of classifiers described, since the majority of the configuration is implemented under the `[edit firewall]` and `[edit interfaces]` hierarchy levels (only the definition of the `forwarding-classes` is implemented under the `[edit class-of-service]` hierarchy level):

```
firewall {
    family inet {
        filter $filter_name$ {
            term $term_name$ {
                from {
                    $match_conditions$;
                }
                then {
                    forwarding-class $class_name$;
                    $other_actions$;
                }
            }
        }
    }
}
interfaces {
    $interface_name$ {
        unit $unit_id$ {
            family inet {
                filter {
                    input $filter_name$;
                }
            }
        }
    }
}
class-of-service {
    forwarding-classes {
        class $class_name$ queue $queue_number$;
    }
}
```

Here, the MF classifier is implemented as a firewall filter. The action applied to matching traffic is to place it into a specified `forwarding-class`.

This firewall filter is applied directly to the logical interface under the `family inet` or `family inet6` hierarchy as an input filter.

Defining Your Classes

Next, you must define your forwarding-classes and how they are mapped to queues. Junos supports up to 32 forwarding-classes, but the maximum number of queues to which they can be mapped is eight. Clearly, this means that you must have a four-to-one mapping of forwarding-classes to queues. It's possible to differentiate between multiple forwarding-classes in a single queue only based on the drop-profile (WRED) applied to each forwarding-class.

WARNING If there are more than eight forwarding-classes, therefore one or more queues have more than one forwarding-class associated with each, then all forwarding-classes associated with a single queue *must* use the same scheduler.

The simplest approach is to use a one-to-one mapping from a forwarding-class to a queue. Eight forwarding-classes are usually adequate and the examples used in this book focus on that model.

TIP The temptation is always to say, "I have traffic from this application that must go into this forwarding-class, therefore I will name the forwarding-class after the application." It is strongly recommend that you *resist that urge*. It is much better to name the forwarding-classes after behaviors so that there is no confusion when traffic from another application, which requires the same behavior, is placed into the same forwarding-class. For example, if a queue is called *video,* but other traffic requiring moderate latency, very low jitter, and moderately low loss is placed into that queue, it can prove confusing.

A sample configuration template for forwarding-classes:

```
class-of-service {
   forwarding-classes {
      class $class_name$ queue $queue_number$ priority $fabric_priority$;
      queue $queue_num$ $class_name$ priority $fabric_priority$;
   }
}
```

Note that only one of the two configuration mechanisms shown here should be used for all forwarding-classes in any single configuration.

Ingress Policers

It is sometimes necessary to ensure that upstream nodes are sending traffic that complies with the contract in place. In order to ensure that the upstream node is adhering to the contract, it is possible to police traffic coming into the network node. This is particularly useful at the boundary between a customer edge device (CE) and a provider edge device (PE) to ensure that the customer is only sending the agreed volume of traffic (possibly per agreed class, as defined using BA markings).

The Junos operating system provides a number of different models for policing on ingress. The main models are:

- Single-Rate Two-Color Marking
- Single-Rate Three-Color Marking
- Two-Rate Three-Color Marking

Single-Rate Two-Color Marking (Policing)

This uses a single value to define the acceptable traffic rate along with a burst size. Above the defined rate, the traffic is considered to be out-of-contract (Red). Below this rate, the traffic is considered to be in-contract (Green). Out-of-contract traffic can be marked (reclassified) or discarded immediately. A configuration template for Single-Rate Two-Color Marking (Policing) is as follows:

```
firewall {
   policer $policer_name$ {
      if-exceeding {
         bandwidth-limit $PIR$;
         burst-size-limit $burst_size$;
      }
      then $action$;
   }
   family $family_name$ {
      filter $filter_name$ {
         term $term_name$ {
            from {
               $match_conditions$;
            }
            then {
               policer $policer_name$;
            }
         }
      }
   }
```

```
}
interfaces {
    $interface_name$ {
        unit $unit_id$ {
            family $family_name$ {
                filter {
                    input $filter_name$;
                }
            }
        }
    }
    $interface_name$ {
        unit $unit_id$ {
            family $family_name$ {
                policer $policer_name$;
            }
        }
    }
}
```

Single Rate Three-Color Marking (srTCM)

Single-Rate Three-Color Marking uses a committed information rate (CIR), a committed burst size (CBS), and an excess burst size (EBS). Traffic within the CIR is Green, traffic above the CBS but within the EBS is Yellow. Traffic above the EBS is Red. A different packet loss priority (PLP) can be applied to each of the three colors. Green is assigned to low PLP, yellow is medium-high PLP, and red is high PLP. A configuration template for Single-Rate Three-Color Marking is as follows:

```
firewall {
    three-color-policer $tcm_policer_name$ {
        single-rate {
            (color-aware|color-blind);
            committed-information-rate $CIR$;
            committed-burst-size $CBS$;
            excess-burst-size $EBS$;
        }
        logical-interface-policer;
        action {
            loss-priority high then discard;
        }
    }
    family $family_name$ {
        filter $filter_name$ {
            term $term_name$ {
                from {
                    $match_conditions$;
```

```
            }
            then {
               three-color-policer $tcm_policer_name$;
               $other_actions$;
            }
         }
      }
   }
}
interfaces {
   $interface_name$ {
      unit $unit_id$ {
         family $family_name$ {
            filter {
               input $filter_name$;
            }
         }
      }
   }
}
```

Two Rate Three-Color Marking (trTCM)

Two-Rate Three-Color Marking uses a committed information rate (CIR) and a peak information rate (PIR). As with srTCM, this approach results in traffic being placed into one of three colors. In this model, two rates are defined. Traffic within the CIR is Green, traffic between the CIR and PIR is Yellow, and traffic above the PIR is Red. A different packet loss priority (PLP) can be applied to each of the three colors. Green is assigned to low PLP, yellow is medium-high PLP, and red is high PLP.

Both srTCM and trTCM policers can operate in either color-aware or color-blind mode.

In color-aware mode, the policer assumes that all packets have already been metered and marked and takes into account the marking already applied. It can rewrite the PLP to a higher value, but not to a lower value.

In color-blind mode, the policer assumes that no previous metering or marking has occurred and ignores any PLP markings. It sets the value of the PLP based entirely on a local decision.

A configuration template for Two-Rate Three-Color Marking is as follows:

```
firewall {
    three-color-policer $tcm_policer_name$ {
        two-rate {
            (color-aware|color-blind);
            committed-information-rate $CIR$;
            committed-burst-size $CBS$;
            peak-information-rate $PIR$;
            peak-burst-size $PBS$;
        }
        logical-interface-policer;
        action {
            loss-priority high then discard;
        }
    }
    family $family_name$ {
        filter $filter_name$ {
            term $term_name$ {
                from {
                    $match_conditions$;
                }
                then {
                    three-color-policer $tcm_policer_name$;
                    $other_actions$;
                }
            }
        }
    }
}
interfaces {
    $interface_name$ {
        unit $unit_id$ {
            family $family_name$ {
                filter {
                    input $filter_name$;
                }
            }
        }
    }
}
```

Forwarding Table Policy

Forwarding table policy allows the manipulation of the forwarding-class and loss-priority based on information in the forwarding table. It is configured and applied with a similar syntax to a MF classifier.

The configuration of the classifier is achieved using a firewall filter with the action setting the forwarding-class.

The firewall filter is then applied as an input filter in the `[edit forwarding-options family $family$]` hierarchy level like this:

```
firewall {
    family $family_name$ {
        filter $filter_name$ {
            term $term_name$ {
                from {
                    $match_conditions$;
                }
                then {
                    forwarding-class $class_name$;
                    $other_actions$;
                }
            }
        }
    }
}
forwarding-options {
    family $family_name$ {
        filter {
            input $filter_name$;
        }
    }
}
routing-instance $instance_name$ {
    forwarding-options {
        family $family_name$ {
            filter {
                input $fiter_name$;
            }
        }
    }
}
```

The filter allows a classification decision to be applied to all traffic associated with a single routing instance, without having to apply the input filter to all interfaces associated with that routing instance.

Egress Policers

Egress Policers rate limit traffic into the egress queues. They are created by configuring a policer that defines the Peak Information Rate (PIR) and the Committed Burst Size (CBS). The specified action is applied to traffic that exceeds the PIR and CBS.

The policer is then used as an *action* in a firewall filter. Thus, it is possible to apply a filter to a subset of traffic entering a queue by specifying match conditions by which to identify the traffic, and then applying the policer to that traffic only.

Finally, the firewall filter is applied to the relevant interface on the output, like the following:

```
firewall {
   policer $policer_name$ {
      if-exceeding {
         bandwidth-limit $max_bandwidth$;
         burst-size-limit $bytes$;
      }
      then {
         discard;
      }
   }
   family $family_name$ {
      filter $filter_name$ {
         term $term_name$ {
            from {
               $match_conditions$;
            }
            then {
               policer $policer_name$;
               $other_actions$;
            }
         }
      }
   }
}
interfaces {
   $interface_name$ {
      unit $unit_id$ {
         family $family_name$ {
            filter {
               output $filter_name$;
            }
         }
      }
   }
}
```

It is also possible to apply a policer to an entire logical interface. In this case, the traffic entering all the queues on the interface is policed at the specified rate, irrespective of the forwarding-class to which it belongs.

Drop-profiles

RED (or more accurately Weighted RED) is implemented using drop-profiles in Junos. Drop-profiles are used to define a series of points on a graph that represent the proportion of matching traffic that is dropped when the queue in which it is waiting reaches a particular queue depth.

There are two basic approaches to creating a drop-profile. The first approach is to manually define a series of points along the graph. The graph will then be a step graph that rises up to the next drop-probability when the matching fill-level is reached.

The template for the first configuration approach is shown below.

```
class-of-service {
   drop-profiles {
      $drop_profile_name$ {
         fill-level $fill_level_pc$ drop-probability $drop_pc$;
      }
   }
   scheduler $scheduler_name$ {
      drop-profile-map loss-priority $loss_priority$ protocol $protocol$ $drop_
profile_name$;
   }
}
```

TIP On the MX series routers, you can only configure `protocol any` when applying a `drop-profile-map`. In general, it is possible to still have a significant positive impact by applying RED to best effort and any other forwarding-class in which the majority of traffic is likely to be TCP-based. Avoid using any drop-profile-map on any scheduler applied to loss-intolerant classes composed primarily of UDP traffic.

An example of the results of this type of approach is shown in the graph of Figure 3.2. This example only requires two points to be defined, but Junos permits up to 64 points to be defined.

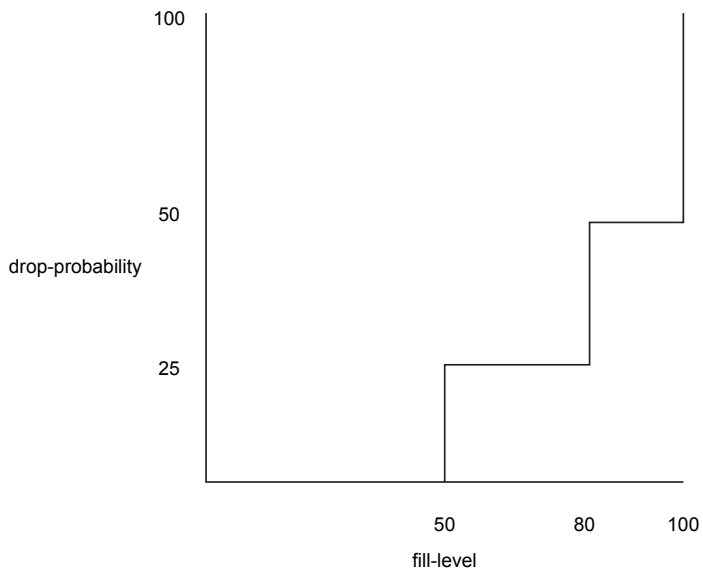


Figure 3.2 Manually Defined Drop-profile Graph

The graph Figure 3.2 would be implemented using the following configuration excerpt:

```
class-of-service {
   drop-profile {
      my_example_drop_profile {
         fill-level 50 drop-probability 25;
         fill-level 80 drop-probability 50;
      }
   }
}
```

The second approach is to define between two and 64 points.

NOTE It is possible to define more than 64 points, but only 64 points will be used to create the drop-profile.

Junos then interpolates between those points to create a "smooth" curve based on 64 discrete points (0, 0) (f1, d1) (f2, d2) (fn, dn) and (100, 100), the first and last points being included by default.

A template for the second approach to defining drop-profiles is given here:

```
class-of-service {
   drop-profiles {
      $drop_profile_name$ {
```

```
        interpolate {
           fill-level [ $fill_1$ $fill_2$ ... $fill_n$ 100 ];
           drop-probability [ $drop_1$ $drop_2$ ... $drop_n$ 100 ];
         }
      }
   }
   schedulers {
      $scheduler_name$ {
         drop-profile-map loss-priority $loss_priority$ protocol $protocol$ $drop_
profile_name$;
      }
   }
}
```

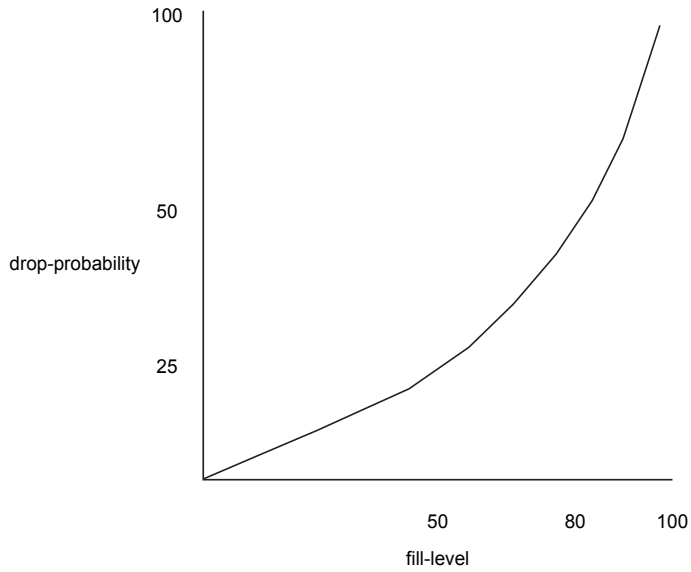


Figure 3.3 Interpolated Drop-profile Graph

The graph in Figure 3.3 can be configured using the following configuration excerpt:

```
class-of-service {
   drop-profiles {
      my_example_interpolate_profile {
         interpolate {
            fill-level [50 80];
            drop-probability [25 50];
         }
      }
   }
}
```

NOTE On the Enhanced Queuing DPCs on the MX series platform, it is only possible to define two points: (f1, 0) and (f2, 100). Below the first point, all matching traffic is transmitted, and above the second point, all matching traffic is discarded, with a straight line drawn between the two values as depicted in Figure 3.4.

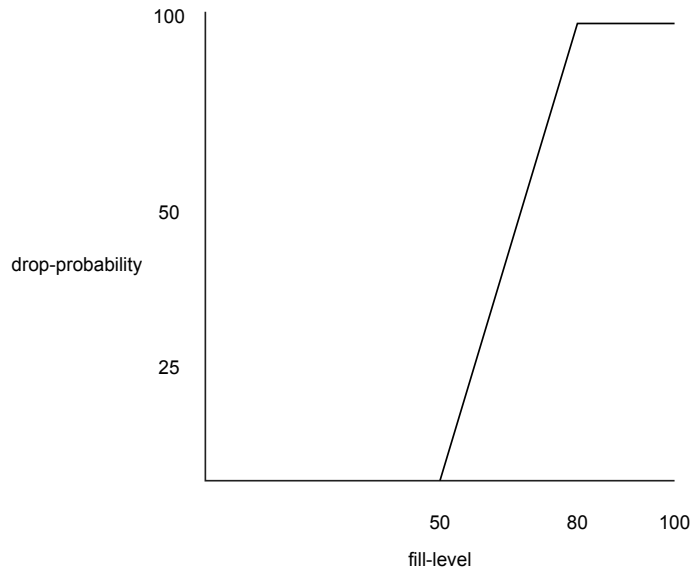


Figure 3.4 Two-point Defined Drop-profile Graph

The graph in Figure 3.4 can be configured using the following configuration excerpt:

```
class-of-service {
   drop-profiles {
      my_example_interpolate_eqdpc_profile {
         interpolate {
            fill-level [50 80];
            drop-probability [0 100];
         }
      }
   }
}
```

Scheduling and Shaping

Scheduling and shaping is possibly the most challenging aspect of QoS to understand. However, it is not impossible if you take a step-by-step approach.

In Junos, scheduling and shaping is configured using the three constructs: schedulers, scheduler-maps, and traffic-control-profiles.

These three elements are built, one upon the other, in order to create a complete definition of the behavior to be applied at the egress interface.

The most basic element is a scheduler, which is used to define the behavior of traffic associated with a single egress queue.

Schedulers are then grouped into a complete set (a scheduler-map), one for each class of traffic, which could possibly be seen on the interface to which the scheduler-map will be (directly or indirectly) applied.

Finally, a traffic-control-profile can be constructed, which merges the functions of the scheduler-map and a shaper. Uniquely, traffic-control-profiles can be applied to various levels of the interface hierarchy: the physical interface, the logical interface, or to an arbitrary group of logical interfaces called an interface-set.

Schedulers

Schedulers are used to configure the behavior of, and the service received by, an individual queue. At this point, the scheduler is an abstract construct that bears no link to any particular forwarding class, so it can be applied to multiple queues without requiring a unique configuration if those queues all require the same behavior.

A scheduler simply defines the size of the buffers, the rate at which the queue is serviced (defined either as a proportion of the total available resource or as an absolute transmission rate), the priority applied to the queue (in the context of the strict priority queuing approach) and the drop-profile that should be associated with this queue. A configuration example would be similar to the following:

```
class-of-service {
   schedulers {
      $scheduler_name$ {
         buffer-size [percent|temporal] $buffer_size$;
         transmit-rate [percent] $transmit_rate$ [rate-limit|exact];
         priority $scheduler_priority$;
```

```
            excess-priority $scheduler_priority_excess$;
            excess-rate [percent] $transmit_rate$;
            drop-profile-map loss-priority $loss_priority$ protocol any drop-profile
$drop_profile_name$;
        }
    }
}
```

Junos implements scheduling in a Priority Queueing – Deficit Weighted Round Robin (PQ-DWRR) model. This makes use of two of the attributes configured above to define the order in which queues are scheduled.

`Priority`, has one of five values: strict-high, high, medium-high, medium-low, and low. In reality, these are four values because strict-high and high behave identically except that strict-high never goes "out of contract."

This is where we need to understand the second important attribute, `transmit-rate`, which defines the "weight" of the queue. This declares how much traffic will be considered "in contract." Once that limit is exceeded, traffic is considered "out of contract."

Queues are scheduled in the following order based on these two attributes.

1. All "in contract" high (and strict-high) priority queues are serviced until they are either all empty or all "out of contract."
2. All "in contract" medium-high priority queues are serviced until they are either all empty or all "out of contract."
3. All "in contract" medium-low priority queues are serviced until they are either all empty or all "out of contract."
4. All "in contract" low priority queues are serviced until they are all empty or all "out of contract."
5. All "out of contract" high priority queues are serviced until they are empty.
6. All "out of contract" medium-high priority queues are serviced until they are empty.
7. All "out of contract" medium-low priority queues are serviced until they are empty.
8. All "out of contract" low priority queues are serviced until they are empty.

NOTE At each stage, after every packet is transmitted, a check is made to ensure that no higher priority packet is awaiting transmission. If such a packet exists, then the scheduler returns to that queue to schedule it. Thus, a high priority queue that is "in contract" will never have to wait more than the time taken to transmit one maximum sized packet before it can be scheduled again.

It is possible to modify the priorities and transmit-rates associated with a queue when it goes out of contract by modifying the `excess-priority` and `excess-transmit-rate`.

NOTE It's necessary to build a scheduler for each behavior that you use. This can lead to a large number of schedulers, particularly on the edge routers, if you have a large variety of interface types and speeds and a large variety of service offerings. Careful definition of the service offerings can dramatically reduce this number without substantively changing the offering.

Scheduler-maps

Scheduler-maps are used to group together a complete set of schedulers and apply them to each of the forwarding-classes that are present on an interface. In creating the scheduler-map, you identify the amount of service assigned to each forwarding-class and the relative priorities of those forwarding-classes.

NOTE If you have created more than eight forwarding-classes and have mapped multiple forwarding-classes to a single queue, it is absolutely mandatory to have all forwarding-classes that map to a single queue use the same scheduler. Traffic in a single forwarding-class can be differentiated based upon the loss-priority by applying a unique drop-profile to each of the loss-priorities in the scheduler.

NOTE It is strongly advised to ensure that every forwarding-class for which any traffic may appear on an interface to which the scheduler-map is applied has a corresponding scheduler. If no scheduler is identified for a forwarding-class, and traffic arrives on that interface for that class, it will receive no explicitly configured service (for example, no buffers, no transmit-rate) and will therefore suffer very poor service unless the interface is completely unused by other traffic.

A configuration template for a scheduler-map is as follows:

```
class-of-service {
   scheduler-maps {
      $scheduler_map_name$ {
         forwarding-class $class_name$ scheduler $scheduler_name$;
      }
   }
   interfaces {
      $interface_name$ {
         unit $unit_id$ {
            scheduler-map $scheduler_map_name$;
         }
      }
   }
}
```

Scheduler-maps are applied either directly to a logical interface (`unit $unit_id$`), as in the template above, or, in the MX series routers, using traffic-control-profiles (see the next section).

When applied directly to the interface, there can be no shaping applied. Policing can, of course, be applied using a standard egress firewall filter.

Traffic-control-profiles

Traffic-control-profiles permit the creation of hierarchical shapers and schedulers. Traffic-control-profiles can be applied at each of the four levels of shaping and scheduling and can be used to apply shapers, schedulers, or both, such as the following:

```
class-of-service {
   traffic-control-profiles {
      $tcp_name$ {
         scheduler-map $scheduler_map_name$;
         shaping-rate $pir$;
         guaranteed-rate $cir$;
      }
      $per_priority_tcp_name$ {
         shaping-rate-priority-high $pir_high$;
         shaping-rate-priority-medium $pir_medium$;
         shaping-rate-priority-low $pir_low$;
      }
   }
   interfaces {
```

```
        interface-set $interface_set_name$ {
            output-traffic-control-profile $per_priority_tcp_name$;
        }
        $interface_id$ {
            output-traffic-control-profile $tcp_name$;
            unit $unit_id$ {
                output-traffic-control-profile $tcp_name$;
            }
        }
    }
}
```

Rewrite Rules

In order to permit downstream nodes to perform classification based on Behavior Aggregates, it is necessary to mark the packets on egress (if they have not already been marked).

Marking can be performed on the same set of address families as can classification (IEEE 802.1p, MPLS EXP bits, IP precedence or *IP DSCP*, or the IPv6 DSCP).

The mechanism for applying a marking is very similar to the reverse of classification. It should be noted, however, that all packets that have a single forwarding-class and loss-priority pair must share the same marking as shown here:

```
class-of-service {
    rewrite-rules {
        $marking_type$ $rewrite_rule_name$ {
            class $forwarding_class$ {
                loss-priority $loss_priority$ code-point $code_point$;
            }
        }
    }
    interfaces {
        $interface_name$ {
            unit $unit_id$ {
                rewrite-rules {
                    $marking_type$ $rewrite_rule_name$;
                }
            }
        }
    }
}
```

Pulling it All Together

The sections in this chapter appeared in a certain order because that is the order in which they are applied as the packet flows through the network node. A better view of this order is illustrated in Figure 3.5.

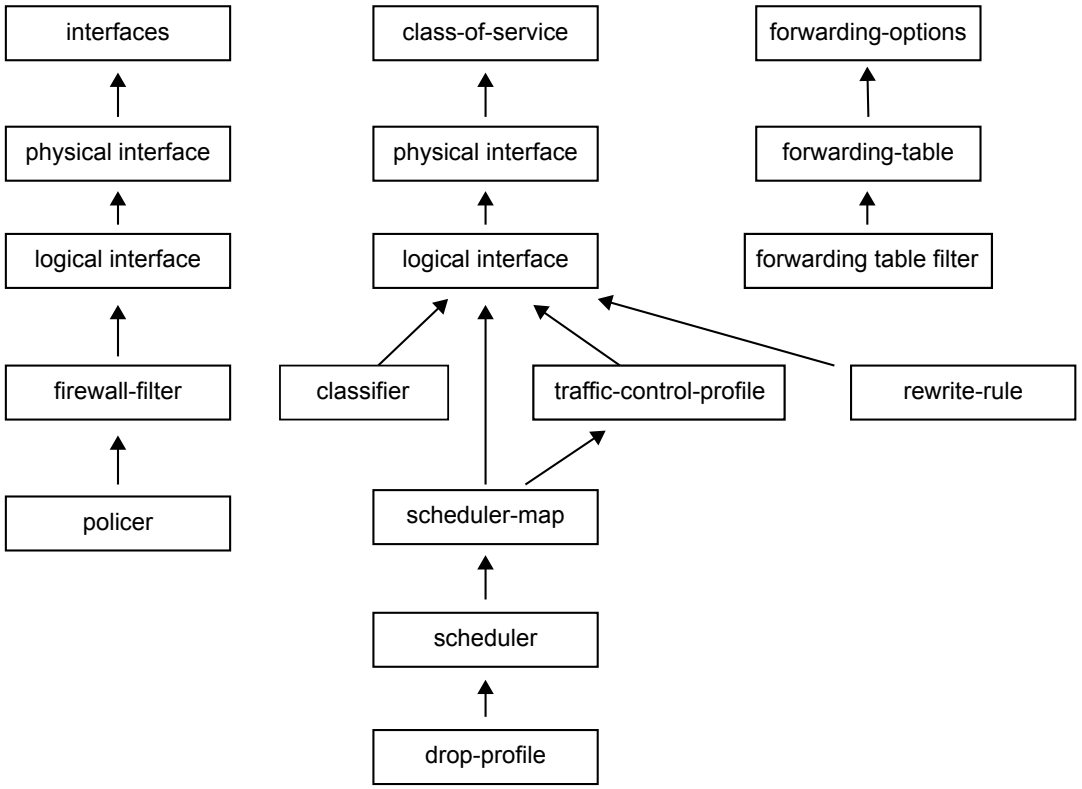


Figure 3.5 Putting it All Together Diagram

Chapter 4

Examples

You should now have a good understanding of the basic verbiage of QoS and how to use the configuration constructs provided in Junos to implement each of the QoS functions. With that as a background, this final chapter provides a few broad examples of how to take the tools presented earlier in the book and create QoS configurations for particular situations.

It's important to remember that as originally stated in Chapter 1, this book is entitled *Day One: Deploying Basic QoS*, emphasing the *basic* nomanclature. It does not aim to be a complete guide to implementing QoS in every possible situation on every possible combination of hardware sold by Juniper Networks since the M40 was released in 1998. The examples herewith are based on the T Series routing platform for the core configuration, the MX Series for the edge and BNG configuration, and the SRX for the security configuration.

Full and detailed configuration guides are available for up-to-date capabilities of each plaform at http://www.juniper.net/techpubs/software/junos.

TIP The identifiers in this chapter's examples that are not part of Junos, but represent tags that will be used to link different elements of the configuration, are depicted in `UPPER_CASE_WITH_UNDERSCORES`. Any keywords in Junos are always in `lower-case-with-dashes`. While it is definitely not a requirement to differentiate variables from keywords in this way, it should provide a very clear separation between the two.

Example of Core QoS Configuration

Here, it is assumed that all traffic passing through this router is MPLS encapsulated and is marked already with a consistent value in the EXP bits of the MPLS header. Note that while this is not entirely realistic, since it is common for some traffic (management traffic, native multicast, etc.) to remain as native IP, it would be necessary to add classifiers and rewrite-rules, specifically for inet-precedence or dscp, in order to apply a class-of-service to that traffic, which would simply add complexity to the configuration example without adding any clarity to the overall design.

It is also assumed that this is running on a router other than an MX (for example, a T Series Router).

```
class-of-service {
    forwarding-classes {
        class NC queue 7 priority high;
        class EF queue 5 priority high;
        class AF11 queue 4 priority low;
        class AF13 queue 3 priority low;
        class AF22 queue 2 priority low;
        class AF42 queue 1 priority low;
        class BE queue 0 priority low;
        class LBE queue 6 priority low;
    }
    classifiers {
        exp BA_CORE_EXP_CLASSIFIER {
            forwarding-class NC {
                loss-priority low code-points 111;
            }
            forwarding-class EF {
                loss-priority low code-points 101;
            }
            forwarding-class AF11 {
                loss-priority low code-points 100;
            }
            forwarding-class AF13 {
                loss-piority high code-points 011;
            }
            forwarding-class AF22 {
                loss-priority medium code-points 010;
            }
            forwarding-class AF42 {
                loss-priority medium code-points 110;
            }
            forwarding-class BE {
                loss-priority high code-points 000;
            }
            forwarding-class LBE {
                loss-priority high code-points 001;
            }
        }
    }
    schedulers {
        HIGH_5PC_5PC_SCHEDULER {
            transmit-rate percent 5;
            buffer-size percent 5;
            priority high;
        }
        HIGH_50PC_RL_20MS_SCHEDULER {
            transmit-rate percent 50 rate-limit;
            buffer-size temporal 20000;
            priority high;
        }
        MEDIUM_HIGH_10PC_20PC_SCHEDULER {
```

```
            transmit-rate percent 10;
            buffer-size percent 20;
            priority medium-high;
        }
        MEDIUM_HIGH_10PC_10PC_SCHEDULER {
            transmit-rate percent 10;
            buffer-size percent 10;
            priority medium-high;
            drop-profile-map loss-priority high protocol any drop-profile AGGRESSIVE;
        }
        MEDIUM_LOW_10PC_10PC_SCHEDULER {
            transmit-rate percent 10;
            buffer-size percent 10;
            priority medium-low;
            drop-profile-map loss-priority high protocol any drop-profile MODERATE;
        }
        LOW_5PC_20PC_SCHEDULER {
            transmit-rate percent 10;
            buffer-size percent 20;
            priority low;
            drop-profile-map loss-priority high protocol any drop-profile AGGRESSIVE;
        }
        LOW_REM_REM_SCHEDULER {
            transmit-rate remainder;
            buffer-size remainder;
            priority low;
            drop-profile-map loss-priority any protocol any drop-profile AGGRESSIVE;
        }
    }
    scheduler-maps {
        CORE_UPLINK_SCHED_MAP {
            class NC scheduler HIGH_5PC_5PC_SCHEDULER;
            class EF scheduler HIGH_50PC_RL_20MS_SCHEDULER;
            class AF11 scheduler MEDIUM_HIGH_10PC_20PC_SCHEDULER;
            class AF13 scheduler MEDIUM_HIGH_10PC_10PC_SCHEDULER;
            class AF22 scheduler MEDIUM_LOW_10PC_10PC_SCHEDULER;
            class AF42 scheduler MEDIUM_LOW_10PC_10PC_SCHEDULER;
            class BE scheduler LOW_10PC_10PC_SCHEDULER;
            class LBE scheduler LOW_REM_REM_SCHEDULER;
        }
    }
    drop-profiles {
        AGGRESSIVE {
            interpolate {
                fill-level [25 60 80];
                drop-probability [40 80 90];
            }
        }
```

```
        MODERATE {
            interpolate {
                fill-level [50 75 95];
                drop-probability [10 25 40];
            }
        }
    }
    rewrite-rules {
        exp CORE_EXP_REWRITE {
            forwarding-class NC {
                loss-priority low code-point 111;
            }
            forwarding-class EF {
                loss-priority low code-point 101;
            }
            forwarding-class AF11 {
                loss-priority low code-point 100;
            }
            forwarding-class AF13 {
                loss-priority high code-point 011;
            }
            forwarding-class AF22 {
                loss-priority medium code-point 010;
            }
            forwarding-class AF42 {
                loss-priority medium code-point 110;
            }
            forwarding-class BE {
                loss-priority high code-point 000;
            }
            forwarding-class LBE {
                loss-priority high code-point 001;
            }
        }
    }
    interfaces xe-0/0/0 {
        unit 0 {
            scheduler-map CORE_UPLINK_SCHED_MAP;
            classifiers {
                exp BA_CORE_EXP_CLASSIFIER;
            }
            rewrite-rules {
                exp CORE_EXP_REWRITE;
            }
        }
    }
}
```

Example of a Distribution/Provider Edge Hierarchical QoS Configuration

Here, at the edge of the network, QoS configuration is often complex, accounting for particular services and combinations of services, in addition to the configurations similar to those required for the core on the uplinks.

This is where hierarchical QoS is usually applied in interfaces over which multiple subscribers, grouped into multiple subsets, are attached.

```
firewall {
    three-color-policer 2M_VOICE_SERVICE_POLICER {
        two-rate {
            color-aware;
            committed-information-rate 2m;
            committed-burst-size 5k;
            peak-information-rate 2500k;
            peak-burst-size 5k;
        }
        action {
            loss-priority high then discard;
        }
    }
    family inet {
        filter UNTRUSTED_SUBSCRIBER_BLEACH_CLASSIFIER {
            term BLEACH_DSCP {
                then {
                    forwarding-class BE;
                    next term;
                }
            }
        }
        filter UNTRUSTED_SUBSCRIBER_MF_CLASSIFIER {
            term VOICE_TRAFFIC_IN {
                from {
                    $match_criteria_for_voice_service$;
                }
                then {
                    forwarding-class EF;
                    three-color-policer 2M_VOICE_SERVICE_POLICER;
                    accept;
                }
            }
            term RECLASSIFY_OUT_OF_CONTRACT_VOICE {
                from {
                    $match_criteria_for_voice_service$;
                    loss-priority high;
```

```
            }
            then {
               forwarding-class BE;
            }
         }
         term EVERYTHING_ELSE_IN {
            then {
               forwarding-class BE;
               accept;
            }
         }
      }
   }
}
interfaces {
   xe-7/0/0 {
      description “Shared Subscriber 10GE – Static Config”;
      unit 1000 {
         family inet {
            filter {
               input {
                  UNTRUSTED_SUBSCRIBER_MF_CLASSIFIER;
               }
            }
         }
      }
      unit 2000 {
         family inet {
            filter {
               input {
                  UNTRUSTED_SUBSCRIBER_BLEACH_CLASSIFIER;
               }
            }
         }
      }
   }
}
class-of-service {
   forwarding-classes {
      class NC queue 7 priority high;
      class EF queue 5 priority high;
      class AF11 queue 4 priority medium-high;
      class AF13 queue 3 priority medium-high;
      class AF22 queue 2 priority medium-low;
      class AF42 queue 1 priority medium-low;
      class BE queue 0 priority low;
      class LBE queue 6 priority low;
   }
   classifiers {
      exp BA_CORE_EXP_CLASSIFIER {
         forwarding-class NC {
```

```
            loss-priority low code-points 111;
        }
        forwarding-class EF {
            loss-priority low code-points 101;
        }
        forwarding-class AF11 {
            loss-priority low code-points 100;
        }
        forwarding-class AF13 {
            loss-piority high code-points 011;
        }
        forwarding-class AF22 {
            loss-priority medium code-points 010;
        }
        forwarding-class AF42 {
            loss-priority medium code-points 110;
        }
        forwarding-class BE {
            loss-priority high code-points 000;
        }
        forwarding-class LBE {
            loss-priority high code-points 001;
        }
    }
    dscp BA_TRUSTED_SUB_DSCP_CLASSIFIER {
        forwarding-class EF {
            loss-priority low code-points 101;
        }
        forwarding-class AF11 {
            loss-priority low code-points 100;
        }
        forwarding-class AF13 {
            loss-piority high code-points 011;
        }
        forwarding-class AF22 {
            loss-priority medium code-points 010;
        }
        forwarding-class AF42 {
            loss-priority medium code-points 110;
        }
        forwarding-class BE {
            loss-priority high code-points 000;
        }
    }
    ieee-dot1p BA_TRUSTED_DSLAM_DOT1P_CLASSIFIER {
        forwarding-class NC {
            loss-priority low code-points 111;
        }
        forwarding-class EF {
            loss-priority low code-points 101;
```

```
        }
        forwarding-class AF11 {
            loss-priority low code-points 100;
        }
        forwarding-class AF13 {
            loss-piority high code-points 011;
        }
        forwarding-class AF22 {
            loss-priority medium code-points 010;
        }
        forwarding-class AF42 {
            loss-priority medium code-points 110;
        }
        forwarding-class BE {
            loss-priority high code-points 000;
        }
    }
}
schedulers {
    HIGH_5PC_5PC_SCHEDULER {
        transmit-rate percent 5;
        buffer-size percent 5;
        priority high;
    }
    HIGH_50PC_RL_20MS_SCHEDULER {
        transmit-rate percent 50 rate-limit;
        buffer-size temporal 20000;
        priority high;
    }
    MEDIUM_HIGH_10PC_20PC_SCHEDULER {
        transmit-rate percent 10;
        buffer-size percent 20;
        priority medium-high;
    }
    MEDIUM_HIGH_10PC_10PC_SCHEDULER {
        transmit-rate percent 10;
        buffer-size percent 10;
        priority medium-high;
        drop-profile-map loss-priority high protocol any drop-profile AGGRESSIVE;
    }
    MEDIUM_LOW_10PC_10PC_SCHEDULER {
        transmit-rate percent 10;
        buffer-size percent 10;
        priority medium-low;
        drop-profile-map loss-priority high protocol any drop-profile MODERATE;
    }
    LOW_5PC_20PC_SCHEDULER {
        transmit-rate percent 10;
        buffer-size percent 20;
        priority low;
```

```
            drop-profile-map loss-priority high protocol any drop-profile AGGRESSIVE;
        }
        LOW_REM_REM_SCHEDULER {
            transmit-rate remainder;
            buffer-size remainder;
            priority low;
            drop-profile-map loss-priority any protocol any drop-profile AGGRESSIVE;
        }
    }
    scheduler-maps {
        CORE_UPLINK_SCHED_MAP {
            class NC scheduler HIGH_5PC_5PC_SCHEDULER;
            class EF scheduler HIGH_50PC_RL_20MS_SCHEDULER;
            class AF11 scheduler MEDIUM_HIGH_10PC_20PC_SCHEDULER;
            class AF13 scheduler MEDIUM_HIGH_10PC_10PC_SCHEDULER;
            class AF22 scheduler MEDIUM_LOW_10PC_10PC_SCHEDULER;
            class AF42 scheduler MEDIUM_LOW_10PC_10PC_SCHEDULER;
            class BE scheduler LOW_10PC_10PC_SCHEDULER;
            class LBE scheduler LOW_REM_REM_SCHEDULER;
        }
    }
    drop-profiles {
        AGGRESSIVE {
            interpolate {
                fill-level [25 60 80];
                drop-probability [40 80 90];
            }
        }
        MODERATE {
            interpolate {
                fill-level [50 75 95];
                drop-probability [10 25 40];
            }
        }
    }
    rewrite-rules {
        exp CORE_EXP_REWRITE {
            forwarding-class NC {
                loss-priority low code-point 111;
            }
            forwarding-class EF {
                loss-priority low code-point 101;
            }
            forwarding-class AF11 {
                loss-priority low code-point 100;
            }
            forwarding-class AF13 {
                loss-priority high code-point 011;
            }
            forwarding-class AF22 {
                loss-priority medium code-point 010;
```

```
    }
    forwarding-class AF42 {
        loss-priority medium code-point 110;
    }
    forwarding-class BE {
        loss-priority high code-point 000;
    }
    forwarding-class LBE {
        loss-priority high code-point 001;
    }
}
dscp SUBSCRIBER_DSCP_REWRITE {
    forwarding-class NC {
        loss-priority low code-point cs7;
    }
    forwarding-class EF {
        loss-priority low code-point ef;
    }
    forwarding-class AF11 {
        loss-priority low code-point af11;
    }
    forwarding-class AF13 {
        loss-priority high code-point af13;
    }
    forwarding-class AF22 {
        loss-priority medium code-point af22;
    }
    forwarding-class AF42 {
        loss-priority medium code-point af42;
    }
    forwarding-class BE {
        loss-priority high code-point be;
    }
    forwarding-class LBE {
        loss-priority high code-point 001000;
    }
}
ieee-dot1p MSAN_DOT1P_REWRITE {
    forwarding-class NC {
        loss-priority low code-point 111;
    }
    forwarding-class EF {
        loss-priority low code-point 101;
    }
    forwarding-class AF11 {
        loss-priority low code-point 100;
    }
    forwarding-class AF13 {
        loss-priority high code-point 011;
    }
    forwarding-class AF22 {
```

```
            loss-priority medium code-point 010;
         }
         forwarding-class AF42 {
            loss-priority medium code-point 110;
         }
         forwarding-class BE {
            loss-priority high code-point 000;
         }
         forwarding-class LBE {
            loss-priority high code-point 001;
         }
      }
   }
   interfaces {
      xe-0/0/0 {
         unit 0 {
            scheduler-map CORE_UPLINK_SCHED_MAP;
            classifiers {
               exp BA_CORE_EXP_CLASSIFIER;
            }
            rewrite-rules {
               exp CORE_EXP_REWRITE;
            }
         }
      }
      xe-7/0/0 {
         unit * {
            rewrite-rules {
               dscp SUBSCRIBER_DSCP_REWRITE;
               ieee-dot1p MSAN_DOT1P_REWRITE;
            }
         }
      }
   }
}
```

Example of Broadband Subscriber Dynamic QoS Configuration

Here, broadband subscriber systems introduce another twist to the challenge of configuring QoS. The subscriber interfaces are dynamic in nature. They only exist when the subscriber is connected and a single logical interface (unit) identifier may be reused for many different subscribers over the lifetime of a Broadband Network Gateway (BNG).

This means that QoS attributes must be dynamically assigned to a subscriber at the time they are connected (and may need to be changed during the lifetime of a single connection). In addition, while it may be possible to define QoS templates, which contain all the values for a particular subset of subscribers, it may also be desirable to define

templates such that attributes are also passed down from the RADIUS server to the BNG when the user is being authenticated and authorized.

This parameterization of the QoS attributes provides a highly flexible mechanism for individual QoS configurations for each subscriber.

NOTE While it is possible to have completely unique QoS configurations per subscriber, it is not recommended. Such an approach would introduce incredible complexity to the system, making the design very difficult to understand and troubleshoot. Instead, it's recommended that a relatively small number of combinations representing each of the service offerings be created, and that those combinations be applied to all subscribers.

```
firewall {
    family inet {
        filter $input_filter_name$ {
            term $term_name$ {
                from {
                    $match_conditions$;
                }
                then {
                    forwarding-class $class_name$;
                    $other_actions$;
                    accept;
                }
            }
        }
        filter $output_filter_name$ {
            term $term_name$ {
                from {
                    $match_conditions$;
                }
                then {
                    forwarding-class $class_name$;
                    $other_actions$;
                    accept;
                }
            }
        }
    }
}
dynamic-profiles {
    $dynamic_profile_name$ {
        predefined-variable-defaults {
            $variable$ $attributes_and_values$;
        }
        interfaces {
```

```
            interface-set “$junos-interface-set-name” {
                interface demux0 {
                    unit “$junos-interface-unit”;
                }
            }
            demux0 {
                unit “$junos-interface-unit” {
                    demux-options {
                        underlying-interface “$junos-underlying-interface”;
                    }
                    family inet {
                        demux-source {
                            $junos-subscriber-ip-address;
                        }
                        filter {
                            input “$junos-input-filter”;
                            output “$junos-output-filter”;
                        }
                        unnumbered-address lo0.0 preferred-source-address $preferred_
address$;
                    }
                }
            }
        }
        class-of-service {
            traffic-control-profiles {
                $tcp_template_name$ {
                    scheduler-map “$junos-cos-scheduler-map”;
                    shaping-rate “$junos-cos-shaping-rate”;
                }
            }
            interfaces {
                demux0 {
                    unit “$junos-interface-unit” {
                        output-traffic-control-profile $tcp_template_name$;
                        rewrite {
                            ieee-802.1 $rewrite_rule_name$;
                        }
                    }
                }
            }
        }
        scheduler-maps {
            $scheduler_map_template_name$ {
                forwarding-class $class_name$ scheduler $scheduler_name$;
            }
        }
        schedulers {
            $scheduler_name$ {
                transmit-rate “$junos-cos-scheduler-tx” exact;
                buffer-size temporal “$junos-cos-scheduler-bs”;
```

```
            priority $scheduler_priority$;
        }
    }
  }
}
```

To make this "template" configuration work, it is necessary to return Juniper Networks Vendor Specific Attributes in a RADIUS Access-Accept. An example configuration for a RADIUS server would be as below:

```
Per User RADIUS VSAs required **example**

Jnpr-CoS-Parameter-Type    T01 BB_SUB_COS
Jnpr-CoS-Parameter-Type    T02 20m
Jnpr-CoS-Scheduler-Pmt-Type    EF T01 2m
Jnpr-CoS-Scheduler-Pmt-Type    EF T02 100
Jnpr-CoS-Scheduler-Pmt-Type    AF11 T01 10m
Jnpr-CoS-Scheduler-Pmt-Type    AF11 T02 15
Jnpr-CoS-Scheduler-Pmt-Type    AF42 T01 7m
Jnpr-CoS-Scheduler-Pmt-Type    AF42 T02 35
Jnpr-CoS-Scheduler-Pmt-Type    BE T01 1m
Jnpr-CoS-Scheduler-Pmt-Type    BE T02 40
Unisphere-Egress-Policy-Name    subscriber_output_policy
Unishpere-Ingress-Policy-Name    subscriber_input_policy
Unishpere-Qos-Set-Name    demux-set
```

Example of a High End Security Node QoS Configuration

Here, high-end security devices from the SRX Series of Services Gateways follow a very similar model for QoS configuration to all other Junos devices. There are some items that need to be taken into account in these devices that don't normally occur in non-security oriented devices.

At the ingress to, and egress from, an IPsec VPN tunnel, the DSCP bits are copied by default from the original (tunneled) packet onto the ESP header's DSCP. Therefore, no rewrite-rule is required if the operator wants a transparent operation. If a different marking is required, then a regular rewrite-rule can be applied.

However, after encapsulation/decapsulation, the packets are not placed into the correct forwarding-class (see the following NOTE). Therefore, a MF classifier on egress is required to place the packets back into the correct forwarding-class so they are given the correct behavior.

NOTE At the time of this writing, the SPCs are not QoS aware. Therefore, if the SPC becomes oversubscribed, it drops packets in a way that doesn't conform to the class-of-service configuration.

```
firewall {
    filter $filter_name$ {
        term SEND_TO_EF {
            from {
                dscp ef;
            }
            then {
                log;
                forwarding-class EF;
                accept;
            }
        }
        term SEND_TO_AF11 {
            from {
                dscp cs2;
            }
            then {
                forwarding-class AF11;
                accept;
            }
        }
        term SEND_TO_AF41 {
            from {
                dscp [cs3 cs6];
            }
            then {
                forwarding-class AF41;
                accept;
            }
        }
        term DEFAULT_PERMIT {
            then accept;
        }
    }
}
class-of-service {
    forwarding-classes {
        queue 2 AF11;
        queue 3 AF41;
        queue 1 EF;
        queue 0 BE;
    }
    interfaces {
        reth0 {
            scheduler-map $scheduler_map_name$;
            unit * {
```

```
            rewrite-rules {
                dscp $rewrite_rule_name$;
            }
        }
    }
}
scheduler-maps {
    $scheduler_map_name$ {
        forwarding-class EF scheduler HIGH_50PC_RL_10MS_SCHEDULER;
        forwarding-class AF11 scheduler MEDIUM_HIGH_20PC_20PC_SCHEDULER;
        forwarding-class AF41 scheduler MEDIUM_LOW_20PC_40PC_SCHEDULER;
        forwarding-class BE scheduler LOW_REM_REM_SCHEDULER;
    }
}
schedulers {
    HIGH_50PC_RL_10MS_SCHEDULER {
        transmit-rate percent 50 rate-limit;
        buffer-size temporal 10000;
        priority high;
    }
    MEDIUM_HIGH_20PC_20PC_SCHEDULER {
        transmit-rate percent 20;
        buffer-size percent 20;
        priority medium-high;
        drop-profile-map loss-priority high protocol any drop-profile MODERATE;
    }
    MEDIUM_LOW_20PC_40PC_SCHEDULER {
        transmit-rate percent 20;
        buffer-size percent 40;
        priority medium-low;
    }
    LOW_REM_REM_SCHEDULER {
        transmit-rate remainder;
        buffer-size remainder;
        priority low;
        drop-profile-map loss-priority any protocol any drop-profile AGGRESSIVE;
    }
}
rewrite-rules {
    dscp $rewrite_rule_name$ {
        forwarding-class EF {
            loss-priority low code-point ef;
        }
        forwarding-class AF11 {
            loss-priority low code-point cs2;
        }
        forwarding-class AF41 {
            loss-priority low code-point cs6;
            loss-priority high code-point cs3;
        }
        forwarding-class BE {
```

```
            loss-priority high code-point be;
        }
    }
  }
}
interfaces {
  reth0 {
    vlan-tagging;
    redundant-ether-options {
      redundancy-group 1;
    }
    unit 10 {
      description “trust”;
      vlan-id 10;
      family inet {
        filter {
          output $filter_name$;
        }
        address $ip_address$/$prefix_length$;
      }
    }
    unit 20 {
      description “untrust”;
      vlan-id 20;
      family inet {
        filter {
          output $filter_name$;
        }
        address $ip_address$/$prefix_length$;
      }
    }
  }
}
```

What to Do Next & Where to Go

http://www.juniper.net/dayone

The *Day One* book series is available for free download in PDF format. Select titles also feature a *Copy and Paste* edition for direct placement of Junos configurations. (The library is available in eBook format for iPads and iPhones from the Apple iBookstore, or download to Kindles, Androids, Blackberrys, Macs and PCs by visiting the Kindle Store. In addition, print copies are available for sale at Amazon or www.vervante.com.)

http://www.juniper.net/books

QoS Enabled Networks: Tools and Foundations, by Peter Lundqvist and Miguel Barreiros. This book, by two experts from Juniper Networks, provides an in-depth treatment of the subject from a more theoretical level all the way through to an understanding of the tools available to influence the behaviors, and finally through to the application of those tools.

http://forums.juniper.net/jnet

The Juniper-sponsored J-Net Communities forum is dedicated to sharing information, best practices, and questions about Juniper products, technologies, and solutions. Register to participate in this free forum.

www.juniper.net/techpubs/

Juniper Networks technical documentation includes everything you need to understand and configure all aspects of Junos, including MPLS. The documentation set is both comprehensive and thoroughly reviewed by Juniper engineering.

www.juniper.net/training/fasttrack

Take courses online, on location, or at one of the partner training centers around the world. The Juniper Network Technical Certification Program (JNTCP) allows you to earn certifications by demonstrating competence in configuration and troubleshooting of Juniper products. If you want the fast track to earning your certifications in enterprise routing, switching, or security use the available online courses, student guides, and lab guides.