JUNIPER NETWORKS

AMBASSADOR

DAY ONE: JUNIPER AMBASSADORS' COOKBOOK FOR 2017

The Juniper Ambassadors take on today's top networking issues in this cookbook full of high performance recipes that will save you time, money, and late night cutovers that can go awry.

By Martin Brown, Matt Dinham, Stefan Fouant, Clay Haynes, Nupur Kanoi, Peter Klimai, Said Klundert, Steve Puluka, David Roy, and Nick Ryce

DAY ONE: JUNIPER AMBASSADORS' COOKBOOK FOR 2017

The Juniper Ambassador program recognizes and supports its top community members and the generous contributions they make through sharing their knowledge, passion, and expertise on J-Net, Facebook, Twitter, and other social networks. In their new Day One cookbook, the Juniper Ambassadors take on some of the top support issues and provide clear-cut solutions and frank discussions on how to keep things running. The recipes in this cookbook are meant to provide quick and tested solutions to everyday networking administration issues.

"Day One: Juniper Ambassadors Cookbook 2017 *covers several complex networking problems and solutions, working with technologies such as EVPN, VXLAN, OSPF sham links, centralized route reflectors, scripted automation with Python, segment routing, and BGP flowspec. Using Junos OS configuration stanzas and command line output analysis, each recipe is demonstrated in detail. The format is perfect for network engineers, explaining each problem and then diving deeply into the solutions.*"

Ethan Banks, Co-Founder of Packet Pushers Interactive, http://packetpushers.net

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

- Configure basic QoS on Junos-enabled devices.
- Migrate from a Cisco LNS to vLNS on a Juniper vMX Series.
- Configure BGP to advertise multiple paths to destinations.
- Utilize VXLAN technologies with EVPN signaling.
- Use OSPF as a PE-CE routing protocol in MPLS VPNs.
- Script network regression testing with Junos PyEZ.
- Selectively leak resources (or subnets) between different VPNs.
- Integrate MX Series routers into Arbor Networks SP.
- Migrate your core to centralized route reflecting and segment routing.

Published by Juniper Networks Books

ISBN 978-1936779697

52000

9 781936 779697

JUNIPer NETWORKS

Day One: Juniper Ambassadors' Cookbook for 2017

By Martin Brown, Matt Dinham, Stefan Fouant, Clay Haynes, Nupur Kanoi, Peter Klimai, Said Klundert, Steve Puluka, David Roy, and Nick Ryce

Published by Juniper Networks Books
Technical Reviewer: Nick Ryce
Editor in Chief: Patrick Ames
Copyeditor and Proofer: Nancy Koerbel
Ambassador Program Manager: Julie Wider
Illustrator: Karen Joice

ISBN: 978-1-941441-47-3 (print)
Printed in the USA by Vervante Corporation.

ISBN: 978-1-941441-48-0 (ebook)

Version History: v1, March 2017
2 3 4 5 6 7 8 9 10

This book is available in a variety of formats at: http://www.juniper.net/dayone. Send your suggestions, comments, and critiques by email to dayone@juniper.net.

About the Juniper Ambassadors

The Juniper Ambassadors are global technical/brand advocates that actively participate across Juniper community and social programs. They are a diverse set of network engineers, consultants, and architects who work in the field with Juniper technologies on a daily basis. The Juniper Ambassadors' mission is spreading the word about the power of Juniper Networks to the world's networking and security engineers. Welcome back, Ambassadors.

About the Authors

Martin Brown (Recipe 1) is a Network Security Engineer for a tier 1 service provider based in the UK and is a Juniper Ambassador. Martin started his career in IT over 20 years ago supporting Macintosh computers and in 1999 earned his first certification by becoming an MCP then an MCSE. In the past six years he has progressed to networking, implementing, and supporting network devices in a number of different environments including airports, retail, warehouses and service provider. His knowledge covers a broad range of network device types and network equipment from most of the major vendors including Cisco, F5, Checkpoint, and of course, Juniper.

Matt Dinham (Recipe 2) is an independent consulting Network Engineer/Architect based in the UK, and is a Juniper Ambassador. Matt has over 15 years experience working within Enterprise and Service Provider environments (both public & private sector). He holds several Juniper Certifications and is certified CCIE#16387 (R&S, SP). Matt can be reached on Twitter - @mattdinham.

Stefan Fouant (Recipe 3) is a Sales Engineer with Copper River Information Technology with over 18 years of experience in the Service Provider and network security industries. He holds several patents in the area of DDoS detection and mitigation and is also a co-author of drafts within the IETF DOTS working group relating to standardized signaling of coordinated DDoS attack filtering and mitigation mechanisms. He was the first person globally to achieve all three expert-level Juniper certifications, and was a technical editor of the book *Juniper MX Series* by O'Reilly (2012).

Clay Haynes (Recipe 4) is an IT professional with over 10 years of experience working on servers, firewalls, and networking. He currently works at Twitter as a Senior Network Security Engineer and is a Juniper Ambassador. Clay currently holds the JNCIE-SEC #69 and JNCIE-ENT #492 certifications.

Nupur Kanoi (Recipe 5) Nupur Kanoi is a senior network engineer for a global service provider where she has gained experience in service provider backbone architecture and design. She also holds JNCIE-ENT (#520), JNCIP-SP, and JNCDS-DC certifications. Nupur can be reached on LinkedIn (linkedin.com/in/nupur-kanoi-520) and on Twitter (@ nupur_kanoi). She is currently working towards her JNCIP-DC.

Peter Klimai (Recipe 6) is a Juniper Ambassador and a Juniper Networks certified instructor working at Poplar Systems, a Juniper-Authorized Education Partner in Russia. He is certified JNCIE-SEC #98, JNCIE-ENT #393, and JNCIE-SP #2253 and has several years of experience supporting Juniper equipment for many small and large companies. He teaches a variety of Juniper classes on a regular basis, beginning with introductory level (such as IJOS) and including advanced (such as AJSEC, JAUT and NACC). Peter is enthusiastic about network automation using various tools, as well as network function virtualization.

Said van de Klundert (Recipe 7) is a Dutch networking enthusiast, Juniper Networks Ambassador, network engineer at Interconnect, and content developer at iNET ZERO. Said has spent most of his career on the service-provider side of things and is known to lab-up and write about whatever sparked his interest. He is a father to his son, husband to his wife, and he enjoys long dinners with friends. JNCIE-SP #2573.

Steve Puluka (Recipes 8, 9, & 10) is a Senior Network Engineer with DQE Communications in Pittsburgh, PA. He is part of a service provider team that manages a fiber optic Metro Ethernet, Wavelength, and Internet Services network spanning 3k route miles throughout western PA. He holds a BSEET along with a dozen Juniper Certifications in Service Provider, Security, and Design. He also has certification and extensive experience in Microsoft Windows server, along with strong VMware skills starting with Version 2. He has enjoyed supporting networks for more than 20 years.

David Roy (Recipe 11) is a Senior Support Engineer for Orange. He is involved in many projects based on IP and MPLS technologies. He loves troubleshooting complex routing and switching issues. He is the author of the *The MX Series, 2nd Edition* (2016, O'Reilly) and Juniper Books' *This Week: An Expert Packet Walkthrough on the MX Series 3D*. David is triple JNCIE.

Nick Ryce (Technical Editor) is a Senior Network Architect for a major ISP based in Scotland, and a Juniper Ambassador. Nick has over a decade of experience working within the Service Provider industry and has worked with a variety of vendors including Cisco, Nortel, HP, and Juniper. Nick is currently certified as JNCIE-ENT #232.

After Reading This Book, You'll Be Able To:

- Configure basic QoS on Junos-enabled devices.
- Migrate from a Cisco LNS to vLNS on a Juniper vMX Series.
- Configure BGP to advertise multiple paths to destinations.
- Utilize VXLAN technologies with EVPN signaling.
- Use OSPF as a PE-CE routing protocol in MPLS VPNs.
- Script network regression testing with Junos PyEZ.
- Selectively leak resources (or subnets) between different VPNs.
- Integrate MX Series routers into Arbor Networks SP.
- Migrate your core to centralized route reflecting and segment routing.

Preface

Someone asked me the other day: *What's It Like Being a Juniper Ambassador?* Most people think of the program as some super secret squirrel group made up of sycophantic Juniper lovers who are blind to any other vendors. This is simply not the case. The Ambassador program is made up of lots of individuals who have a great love of all things geek and a deep knowledge of Juniper tech as well as other vendors. Juniper allows us access to product managers and developers to help us get a better understanding of their products, not only to help others in the community, but also to help us make informed decisions about the products we use and recommend.

All of the Juniper Ambassadors are either active on multiple social media outlets or participating/lurking on the J-Net forums to help users with any issues they may have. We are always bouncing ideas off of each other, not only on social media but at the yearly gathering we attend. It's been fun, and how many corporate programs can you claim as fun?

Of course one of the reasons for the networking frolic is our champion and program manager, Julie Wider at Juniper Networks. She supports us when we need it and ignores us when the workplace humor kicks in. Thank you Julie, from all of us.

Which brings me to the following Ambassadorian fine print: Not all of the Ambassadors contributed to this book. There are almost two dozen Ambassadors in all, and some of us have busy day jobs. What you don't see is all of the support and comradery (and cheering) that goes on for those who temporarily have some free time and are willing to spend that time helping other network engineers. So although you see some great names as authors of this cookbook, there are even more network engineers supporting them, reviewing their writing, and testing recipes. Thank you to all Juniper Ambassadors.

You have to be nominated to be a Juniper Ambassador, and during your tenure you are expected to contribute to books, blogs, tweets, and posts. Ambassadors give feedback to Juniper management and to product line managers, and it's real feedback, because they can, and because they work with the product all day long. If you tend to networks, you know how chaotic the job is getting – well part of being an Ambassador is to relay that message to Juniper and to Juniper's customers. That's why this is the third Ambassador *Day One* book. Nobody knows it better than a Juniper Ambassador: *It's Day One and You've Got A Job To Do.*

Enjoy this year's Ambassador Cookbook. It's a good one.

Nick Ryce, March 2017
Technical Reviewer and Cat Herder Extraordinaire,
Day One: Juniper Ambassadors' Cookbook 2017

Recipe 1: Basic QoS in the Junos OS

The enterprise networks of today tend to be large networks carrying many different types of traffic such as email, customer data, web pages, videos, and voice data for telephone calls. This means there is all kinds of data vying for as much bandwidth as possible, not to mention that different types of traffic have different characteristics.

As an example, streaming video can be a continuous stream of high bandwidth data or it can be 'bursty' – sending a lot of data, then stopping for a bit, then sending another lot of high volume data. Alternatively, data traffic such as the type containing plain text emails can be relatively small in size and it doesn't really matter whether it takes 500ms or 10 seconds to reach its destination just as long as it gets there; that's all that matters.

Voice over IP, or VoIP, are telephone calls made via the network and their data stream is relatively small, being around 64Kbps, but the stream is constant, without interruption. Just as important, however, is that the delay between the sending phone sending a packet of data and the recipient phone receiving the packet of data needs to be the same throughout the entire conversation. This is called *delay variation*, more commonly known as *jitter*.

Problem

The problem is, how do you ensure that all the high bandwidth or bursty data doesn't consume your network and prevent other data, like emails, from reaching the email server, while simultaneously making sure that voice data is treated as a priority with very little delay variation? How do you provide some level of control to the data that is traversing your network?

Unsurprisingly, the title of this recipe is a bit of a spoiler –QoS allows us to control how much bandwidth a stream of data is using, determine whether that traffic is becoming more than the network can handle, and also decide how it treats traffic that grows above the bandwidth limit.

At first glance this can seem like a bit of an oxymoron; you are dealing with excessive bandwidth by limiting the bandwidth available to certain traffic streams, and dropping traffic once the limit is reached, however, QoS also allows you to tell your network devices that some traffic is more important than other traffic, and as such should be treated as a priority.

When you put these two main features together you get something interesting. Imagine for a moment that a network link is 10Mb and that the streaming video is 4k high bandwidth and wants to take up 100% of the bandwidth, but in bursts. You can tell your Junos OS-based devices that the video stream is only allowed to take up 50% of the bandwidth and that any data above the limit that has been set would be dropped. In addition, when there is voice traffic waiting to be sent onto the wire, the voice traffic must be sent ahead of the video traffic.

This leads to another question: *How does the Junos OS allow a certain type of traffic to 'overtake' another type of traffic?* Junos OS-based devices don't have multi-lane freeways going through them to the cables, so how do they hold some types of traffic while sending others? The solution is quite brilliant.

When traffic is about to be sent down the wire, the traffic is first placed into a buffer, which temporarily stores the traffic until the wire is free to send the data. For the purposes of QoS, the buffer is divided into up to eight segments. These eight segments are known as *queues;* Junos can monitor these queues and if it sees traffic in one of the queues with a higher priority it will send this traffic first, then send traffic from queues with a lower priority.

In addition, Junos doesn't just keep sending traffic from a high priority queue and forget about the lower priority queues. The Junos OS will send several packets from a high priority queue, then send one or more from the lower priority queues, so that other types of traffic don't feel left out or so that other traffic isn't starved of bandwidth while a phone call takes place.

When it comes to controlling how much bandwidth a type of traffic is allowed to use, there are two options: *policing* or *shaping*. Policing is harsh in that traffic over the limit set is dropped. Shaping, on the other hand, treats data slightly differently from policing in that when data of a bursty nature is being sent there is a period when a lot of data is being sent and then a period when no data is sent. In this instance, instead of dropping excess data, shaping holds onto the data and sends it between data bursts. Figure 1.1 shows an example of policing that drops data exceeding 50%. The data in this graph is dark in color, and, as you can clearly see, any data in the shaded area will be dropped.

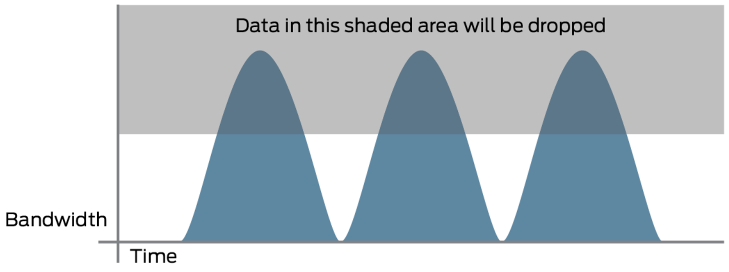


Figure 1.1 Policing Data at 50%

Figure 1.2 shows a different story. In this case, there is no data above the red line, which means no data has been dropped. Instead, what has happened is that Junos OS has held onto the data and sent it between gaps in the bursts of data. This is indicated by shaded gaps between the data bursts that were previously white.

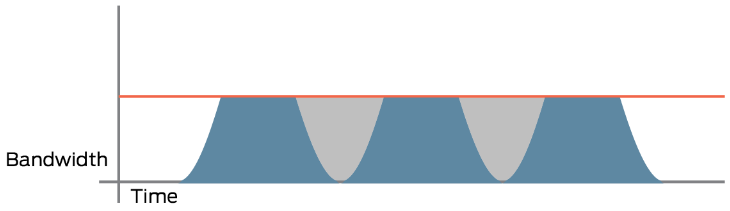


Figure 1.2 Shaping Data at 50%

NOTE While this recipe is meant to be as informative as possible, QoS is a topic worthy its own book, therefore this chapter will cover only the very basics of QoS and readers wishing to study it in more depth may find the *Day One* book *Day One: Deploying Basic QoS* by Guy Davies, a useful addition to their digital library: http://www.juniper.net/us/en/training/jnbooks/day-one/fundamentals-series/deploying-basic-qos/ .

Having given the basics of how QoS works, let's look at a real-world scenario and how to implement QoS on an enterprise network. In this instance, ACME Company is looking to add VoIP telephones to their network and will need QoS implementing in order to provide the best possible service.

The first thing you need to bear in mind is that while ACME has a large enterprise network, attempting to describe and configure the entire network in this chapter would probably end up doubling the size of this book. So instead of describing the entire network, let's focus on a small area. Figure 1.3 illustrates the topology of the ACME network.

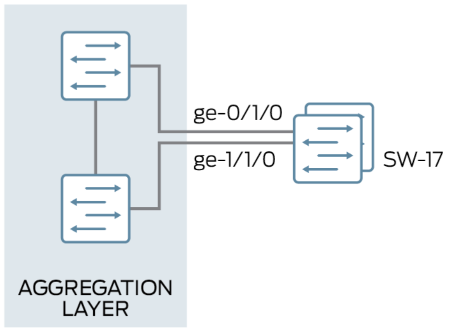


Figure 1.3 ACME's Enterprise Network

In the scenario you will configure a switch in the access layer, SW-17, to allow for implementing a new VoIP service. SW-17 is a Juniper EX Series switch configured as a virtual chassis with two members. This switch is connected to the aggregation layer via ports ge-0/1/0.0 and ge-1/1/0.0

The first thing you need to do is configure the switch ports so you can give your new VoIP phones connectivity. Originally, this was a fairly straightforward task, as the port would be configured as an access port and was made a member of whatever VLAN was used to carry voice data, so you ended up with a topology similar to what we see in Figure 1.4.

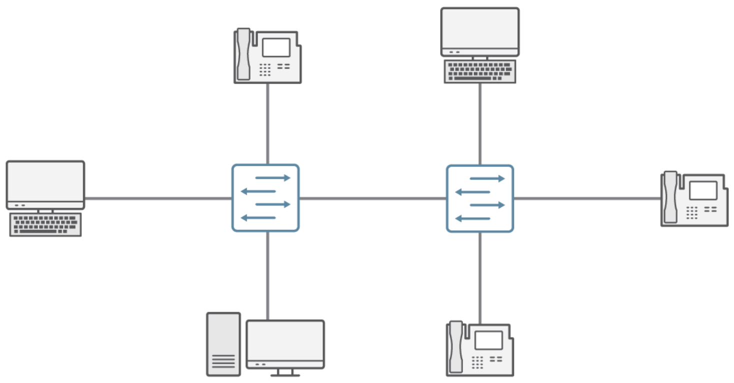


Figure 1.4 Adding VoIP Telephones to a LAN

There was, however, one small issue with this design. Imagine a place like a call center or a contact center where every agent has their own phone and workstation. If the call center was using one of the older style PABX (Private Automatic Branch Exchange) systems, each phone was plugged into a socket, which in turn had a cable run to an internal exchange. This cable could have been standard telephone cable; however, I recall installing a new CAT5 network back in 1993, and rather than running separate telephone cables, I connected the PABX internal exchange to the telephones via the same CAT5 cable.

But what if the call center's PABX telephones were replaced with VoIP telephones? The number of switch ports required would effectively double and a call center with 100 agents would need 100 ports for the workstations and a further 100 ports for the new VoIP telephones. This means you need to purchase another pair of 48 port switches.

Thankfully, manufacturers of VoIP telephones came up with a brilliant solution – add a port to the back of the telephone so you can connect the workstation. So, in a call center with 100 agents, only 100 ports are required, as the phones and workstations are effectively sharing ports and you end up with a design similar that shown in Figure 1.5.

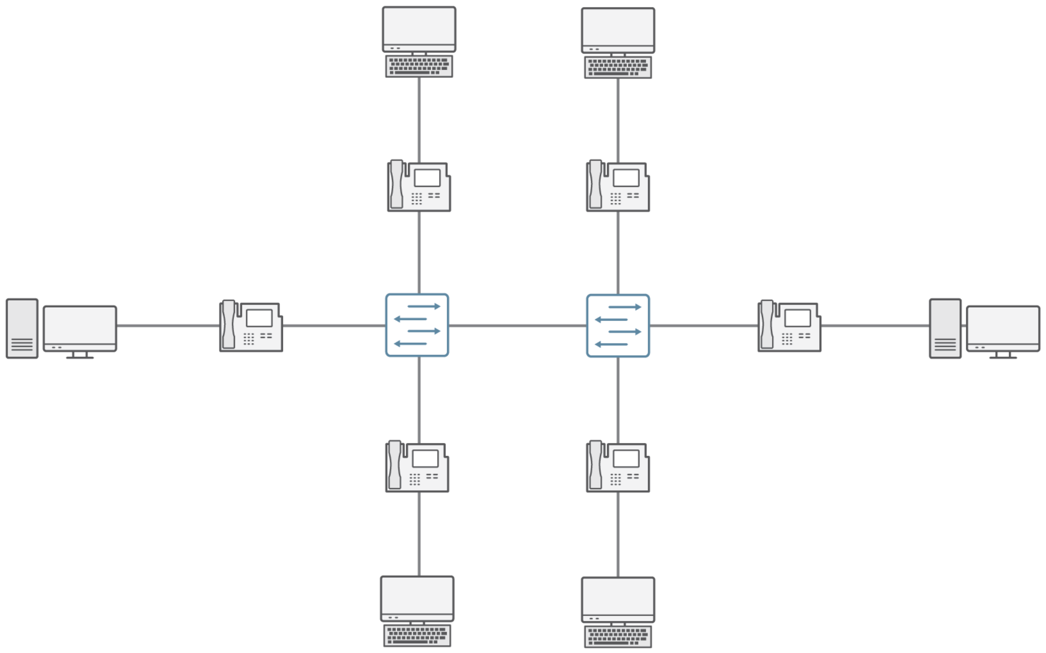


Figure 1.5 Adding Connecting Clients Through VoIP Telephones

Now the ports are receiving data from two different VLANs: the data VLAN for the workstation, and the voice VLAN for the telephone. Therefore, the switch needs some way of differentiating between the two different data streams. The solution to this is to have the phone send frames that are tagged with the voice VLAN, while the workstation continues to send untagged frames.

The phone, of course, needs to be told which VLAN the frame needs to be tagged with, and there are two methods to achieve this: either configure the phone to send frames tagged as the voice VLAN, or use a technology called "LLDP-MED" where the phone communicates with the switch using the Link Layer Discovery Protocol (LLPD) and the switch tells the phone which VLAN is used for the Voice VLAN.

When the switch receives a frame, it checks for the VLAN tag. If the tag is present the switch sends the frame to voice VLAN, and if there is no tag the switch forwards the frame to the data VLAN. This solution, however, means that the switch needs to be configured to deal with tagged frames. Let's do that, otherwise the switch will discard them.

Solution

Let's start by making sure that the relevant VLANs have been created on the switch. If these have not been created, running the following commands creates the VLANs *FINANCE* and *VOICE*:

```
set vlans FINANCE vlan-id 100
set vlans VOICE vlan-id 300
```

Once the VLANs have been created, the next step is adding the switch port to the correct VLAN. In this case, interface ge-0/0/10.0 is made a member of the VLAN FINANCE:

```
set interfaces ge-0/0/10.0 family ethernet-switching vlan members FINANCE
```

Another option is setting the port as a trunk link and setting the native VLAN as the data VLAN, so when the client sends an untagged frame the switch knows that this frame is for the data VLAN. Let's use a port set as an access port; now all you need to do is tell the EX Series that if any frames tagged as belonging to the VOICE VLAN are received on port ge-0/0/10.0 they are carrying VoIP traffic:

```
set ethernet-switching-options voip interface ge-0/0/10.0 vlan VOICE
```

Once the `interface` and `ethernet-switching-options` configuration has been committed to the EX's running configuration it is ready for a VoIP phone to be connected, except now the traffic from the phones is treated as ordinary traffic.

As a result, you need to tell the switch to treat the VoIP traffic *differently* from data traffic and that is done using something called *Class of Service* or CoS. CoS differs from QoS in that CoS operates on EX Series switches and combines the Layer 2 or VLAN level with Layer 3. While QoS operates exclusively at Layer 3, it can identify traffic marked as a priority by CoS and treat it as such.

First you need to help the switch identify how important a traffic stream is. This is done by marking the frames in a special field within the frame header or IP header. In a frame, this field is known as the *802.1Q* or *Tag* field, and is commonly used for VLAN tagging, but three bits of it are reserved specifically for CoS markings.

There are in fact two ways a frame can be marked: the phone can send the frame with the marking already applied, or the EX Series can be configured to mark the frame as it's received and to remove the marking if one has already been applied. In this case, let's mark the frame with the importance of `assured-forwarding` and, again, this can be achieved by utilizing two methods. The first is to use the `forwarding-class` keyword as follows:

```
set ethernet-switching-options voip interface ge-0/0/10.0 forwarding-class assured-
forwarding
```

What exactly does *assured forwarding* mean? You may recall that QoS uses queues in order to allow some traffic to overtake other traffic. Assured forwarding is just a name for a queue and this queue is used by CoS to allow traffic to overtake other traffic when the EX Series is sending it across a trunk link to another switch, or if the traffic is about to be routed to another subnet. QoS can translate this queue name to a QoS queue name.

The Junos OS refers to the queues as *forwarding classes* and allows for up to eight forwarding classes to be created. By default, however, four CoS forwarding classes are created: *best effort*, *expedited forwarding*, *assured forwarding*, and *network control*. Table 1.1 lists the queue types and describes how Junos OS treats traffic assigned to these queues.

Table 1.1 Junos OS Default CoS Forwarding Classes

Forwarding Class	Abbreviation	How Traffic Is Treated
Best Effort	be	If the bits in the field are left as all 0's, the frame is classed as best effort. This means that the Junos OS does not apply any special CoS handling to packets and that these packets are usually the first to be dropped under congested network conditions.
Expedited Forwarding	ef	The Junos OS guarantees a certain amount of bandwidth for packets marked as ef, in addition to assuring low loss, low delay, and low delay variation end-to-end. Junos also allows traffic to exceed the bandwidth but packets exceeding the allowed bandwidth may be forwarded out of sequence or dropped.
Assured Forwarding	af	Junos OS will do its absolute best to assure that traffic marked as af will reach its destination with the minimum of delay and without any discards, as long as the traffic stays within the bandwidth limit. Excess traffic is permitted but a tail drop may apply to excess bandwidth in times of congestion.
Network Control	nc	Frames marked as nc are typically used to send traffic from dynamic routing protocols, and as such, failure to receive them could have an adverse effect on network connectivity. Therefore, it is highly likely that these packets will never be dropped but nc packets could be delayed.

If one wanted to see which forwarding classes were added to the switch then the `show class-of-service forwarding-class` command can be used. The following output shows the default forwarding classes configuration:

```
admin@SW-17> show class-of-service forwarding-class
Forwarding class            ID       Queue    Policing priority    SPU priority
Best-effort                  0           0             normal            low
expedited-forwarding         1           5             normal           low
assured-forwarding           2           1             normal           low
network-control              3           7             normal           low
```

In this recipe, the default forwarding classes will be used, but should you wish to create a new forwarding class it's simply a matter of using a set command, for example, if an administrator wanted to create a queue called *cute-cat-videos*, the command would be:

```
set class-of-service forwarding-classes class cute-cat-videos queue-num 4
```

At the end of this command is the number 4. This is an ID number and each forwarding class should have its own unique ID. It is also a good idea to set the IDs in order of priority, so assuming traffic identified as containing cute cat videos is considered more important than say, VoIP, you could give this forwarding class the ID of 3 and move `network-control` to ID 4.

As mentioned earlier, there is a second method that can also be utilized for the EX Series to mark traffic with its appropriate forwarding class. The second method is to use a *firewall filter*. Firewall filters are typically used for allowing and denying traffic streams, however in this case the filter is used to identify traffic streams and mark them appropriately with different terms used to classify different streams of traffic. Identifying traffic streams using a firewall filter is called a *multifield classifier* as the firewall filter can match against source and destination addresses, protocols and applications.

In this recipe, a firewall filter will be created with the name *COS-FILTER,* and the first term, which will be given the name *VIDEO*, will identify the streaming video traffic that is using the time streaming protocol RSTP on TCP and UDP port 554, after which, the filter will mark the traffic as belonging to the forwarding class `expedited-forwarding` and set the loss-priority, which tells Junos OS whether it can drop packets from the stream during periods of high congestion. In this case, the loss-priority will be set to low, meaning there is a low probability of packets in this stream being dropped during periods of high congestion:

```
edit firewall family ethernet-switching filter COS-FILTER
set term VIDEO from protocol tcp
set term VIDEO from protocol udp
set term VIDEO from source-port 554
set term VIDEO then forwarding-class expedited-forwarding
set term VIDEO then loss-priority low
top
```

The next term, in theory, wouldn't be necessary on a port connected to a client, however let's add it just to show how it's done. This term will be used to identify traffic from routing protocols and other important network protocols and to add it to the forwarding class `network-control`.

In the previous term, the port and protocol were used to identify the traffic. When identifying routing protocols, you can use what is known as the *IP precedence*, which is another form of QoS marking in the IP header. Routing protocols mark the packet with the IP precedence as it is transmitted. You can identify traffic marked with the appropriate IP precedence using a firewall filter, and assign it to the correct forwarding class.

The IP precedence markings that apply here are `net-control` and `internet-control`. A term will be created with the name *NETWORK-PROTOCOLS* and this assigns traffic identified as a routing or network protocol to the forwarding class `network-control`:

```
edit firewall family ethernet-switching filter COS-FILTER
set term NETWORK-PROTOCOLS from precedence net-control
set term NETWORK-PROTOCOLS from precedence internet-control
set term NETWORK-PROTOCOLS then forwarding-class network-control
set term NETWORK-PROTOCOLS then loss-priority low
top
```

Finally, a term must be created to assign all other traffic to the forwarding class `best-effort`. This term will not match against any protocol, port, or IP precedence, and will not include any `from` statement. By leaving out a `from` statement, the filter matches all traffic that hasn't been otherwise matched by a previous term. The term in this case is given the name *OTHER-TRAFFIC*:

```
edit firewall family ethernet-switching filter COS-FILTER
set term OTHER-TRAFFIC then forwarding-class best-effort
set term OTHER-TRAFFIC then loss-priority high
top
```

This firewall filter is now ready to be assigned to the port to which our new VoIP phone was connected, which was port ge-0/0/10.0. This filter needs to identify traffic coming in on that port, therefore the direction is set as `input`:

```
set interfaces ge-0/0/10.0 family ethernet-switching filter input COS-FILTER
```

Once the forwarding classes are created, the next step is to tell the Junos OS what to do with the traffic within those forwarding classes. To achieve this you use something called a *scheduler* to tell Junos how much bandwidth to allocate, and then use a *scheduler map* to link the scheduler to the forwarding class. In this instance, there are four forwarding classes, and although the Junos OS does create some by default, you will create your own. The first scheduler is the one used for traffic in the `best-effort` forwarding class.

One tends to find that most network traffic, such as emails with large attachments or large databases being uploaded to the server, falls into this category. This traffic might be important to an employee or the company, but it is not time sensitive and can be treated with a lower priority, while the bandwidth allocated to this forwarding class should be reasonable.

The scheduler is configured to allocate 60% of the buffer to traffic within this class and the amount of bandwidth the traffic is allowed to use is set to 60%. Finally, the priority is set as low, which means if traffic with a priority set as high enters the buffer, it will probably be sent ahead of traffic within this forwarding class. The scheduler will be given a name of `NOT-IMPORTANT`:

```
set class-of-service schedulers NOT-IMPORTANT buffer-size percent 60
set class-of-service schedulers NOT-IMPORTANT priority low
set class-of-service schedulers NOT-IMPORTANT transmit-rate percent 60
```

The priority can be set with five different levels; low, medium-low, medium-high, high, and strict-high with the probability of the traffic being dropped decreasing as the priority increases. The priority of strict-high may only be assigned to one scheduler in a map. Attempting to assign it to two schedulers will result in the following error:

```
admin@sw-17# commit check
[edit]
  'class-of-service scheduler-maps OUR-COS-MAP'
    More than one scheduler with the priority strict-high for scheduler-map OUR-COS-MAP
error: configuration check-out failed
```

NOTE Traffic labelled as `strict-high` will always be sent ahead of traffic marked as `low`, therefore it is important to assign this priority only to low bandwidth traffic in order to prevent queue starvation.

In this recipe's example, the transmit rate has been set as a percentage, however there is an option to set the transmission rate as bits per second. The Junos OS also allows the use of the `exact` option, which means that is there is a lot of congestion on a link because traffic in a forwarding class linked to a scheduler with the `exact` option set will be guaranteed that transmission rate. As long as the amount of traffic doesn't exceed the transmission rate, traffic in that forwarding class will never be dropped.

The next forwarding class is expedited-forwarding. Traffic in this queue is more important than email but not as important as VoIP traffic. An example of this traffic is streaming video. If traffic is delayed, it could break the stream leading to drop outs or the video going pixelated. In this case, the buffer queue and transmit rate is set to 20% and the priority is set as medium-low meaning this traffic is less likely to be dropped ahead of the NOT-IMPORTANT traffic:

```
set class-of-service schedulers VIDEO-STREAM buffer-size percent 20
set class-of-service schedulers VIDEO-STREAM priority medium-low
set class-of-service schedulers VIDEO-STREAM transmit-rate percent 20
```

The assured-forwarding forwarding class is next, which, as you will recall, was the forwarding class our VoIP traffic was assigned to. This traffic is not expected to be high bandwidth, therefore the transmit rate and buffer percentage is set at 15%. The priority level, however, is set at high, meaning it is unlikely this traffic will be dropped during congestion:

```
set class-of-service schedulers VOIP-SCHEDULER buffer-size percent 15
set class-of-service schedulers VOIP-SCHEDULER priority high
set class-of-service schedulers VOIP-SCHEDULER transmit-rate percent 15
```

The next forwarding class is network-control. This traffic is important because if the traffic flow stopped, routes could be marked as inaccessible and removed from routing tables. Therefore, this traffic is marked as strict-high but as this traffic is expected to be very low bandwidth, the transmit rate and buffer are set at 5%:

```
set class-of-service schedulers NETWORK-CONTROL buffer-size percent 5
set class-of-service schedulers NETWORK-CONTROL priority strict-high
set class-of-service schedulers NETWORK-CONTROL transmit-rate percent 5
```

Once the schedulers are created they can be linked to the forwarding class using a scheduler map. This a simple linking of one to the other. In our case, the scheduler map is given the name of OUR-COS-MAP and in this case, as the command is so long, the changes will be made under the class-of-service scheduler-maps OUR-COS-MAP hierarchy:

```
edit class-of-service scheduler-maps OUR-COS-MAP
set forwarding-class best-effort scheduler NOT-IMPORTANT
set forwarding-class expedited-forwarding scheduler VIDEO-STREAM
set forwarding-class assured-forwarding scheduler VOIP-SCHEDULER
set forwarding-class network-control scheduler NETWORK-CONTROL
top
```

The next step is telling Junos which interfaces use this scheduler map. Without specifying this option, the OS would use the default scheduler map, which you want to override. You want to apply the scheduler map to the interfaces in the aggregation layer, which would be done by using the following commands:

```
set class-of-service interfaces ge-0/1/0 scheduler-map OUR-COS-MAP
set class-of-service interfaces ge-1/1/0 scheduler-map OUR-COS-MAP
```

As mentioned earlier in this recipe, another option is shaping the traffic rather than limiting the transmit rate, which means holding onto traffic in the buffer instead of immediately dropping the traffic and sending the traffic during quiet periods between bursts. This treatment is more useful for video traffic therefore the scheduler *VIDEO-STREAM* has the transmit rate configuration replaced with the setting `shaping-rate`. The shaping rate is set at the same 20%:

```
delete class-of-service schedulers VIDEO-STREAM transmit-rate percent 20
set class-of-service schedulers VIDEO-STREAM shaping-rate percent 20
```

All that remains now is to commit the changes and check to ensure that the scheduler map correctly maps the forwarding classes to the schedule, and that the scheduler map is assigned to the interfaces to the aggregation layer.

To check the configuration of the scheduler map, use the `show class-of-service scheduler-map` command. If you follow the command with the map name, then only details on that particular scheduler map will be displayed, as is done here:

```
admin@sw-17> show class-of-service scheduler-map OUR-COS-MAP

Scheduler map: OUR-COS-MAP, Index: 5401

  Scheduler: NOT-IMPORTANT, Forwarding class: best-effort, Index: 47922
    Transmit rate: 60 percent, Rate Limit: none, Buffer size: 60 percent,
    Buffer Limit: none, Priority: low
    Excess Priority: unspecified
    Drop profiles:
      Loss priority   Protocol    Index    Name
      Low             non-TCP         1    <default-drop-profile>
      Low             TCP             1    <default-drop-profile>
      High            non-TCP         1    <default-drop-profile>
      High            TCP             1    <default-drop-profile>

  Scheduler: VIDEO-STREAM, Forwarding class: expedited-forwarding, Index: 4698
    Transmit rate: unspecified, Rate Limit: none, Buffer size: 20 percent,
    Buffer Limit: none, Priority: medium-low
    Excess Priority: unspecified
    Shaping rate: 20 percent
    Drop profiles:
      Loss priority   Protocol    Index    Name
      Low             non-TCP         1    <default-drop-profile>
      Low             TCP             1    <default-drop-profile>
      High            non-TCP         1    <default-drop-profile>
      High            TCP             1    <default-drop-profile>

  Scheduler: VOIP-SCHEDULER, Forwarding class: assured-forwarding, Index: 4862
    Transmit rate: 15 percent, Rate Limit: none, Buffer size: 15 percent,
    Buffer Limit: none, Priority: high
    Excess Priority: unspecified
    Drop profiles:
      Loss priority   Protocol    Index    Name
      Low             non-TCP         1    <default-drop-profile>
      Low             TCP             1    <default-drop-profile>
      High            non-TCP         1    <default-drop-profile>
      High            TCP             1    <default-drop-profile>

  Scheduler: NETWORK-CONTROL, Forwarding class: network-control, Index: 26790
    Transmit rate: 5 percent, Rate Limit: none, Buffer size: 5 percent,
    Buffer Limit: none, Priority: strict-high
    Excess Priority: unspecified
    Drop profiles:
      Loss priority   Protocol    Index    Name
      Low             non-TCP         1    <default-drop-profile>
      Low             TCP             1    <default-drop-profile>
      High            non-TCP         1    <default-drop-profile>
      High            TCP             1    <default-drop-profile>
```

To check whether the scheduler map has been correctly assigned to the outgoing interface, use the `show class-of-service interface` command to show all interfaces or use the interface name to check specific interfaces, as is done here:

```
root@sw-17> show class-of-service interface ge-0/1/0.0
Physical interface: ge-0/1/0, Index: 133
Queues supported: 4, Queues in use: 4
  Scheduler map: OUR-COS-MAP, Index: 2
  Congestion-notification: Disabled
```

Once a substantial amount of CoS controlled traffic has been flowing over the interface, the `show interfaces ge-0/1/0 extensive` command can be used to view interface statistics, including CoS queue information, however, if you use the pipe option followed by `find Queue counters`, you can see just the CoS statistics:

```
show interfaces ge-0/1/0 extensive | find "Queue counters"
  Queue counters:      Queued packets  Transmitted packets     Dropped packets
    0                            0                5669                 0
    1                            0                   0                 0
    5                            0                   0                 0
    7                            0                1911                 1
  Queue number:        Mapped forwarding classes
    0                 best-effort
    1                 assured-forwarding
    5                 expedited-forwarding
    7                 network-control
```

MORE? Remember that this recipe only scratches the surface of configuring QoS or CoS on a Juniper EX Series switch, and key areas such as *drop probability* have not been covered or discussed. While this recipe can help as an introduction to QoS and possibly any VoIP deployment your organization may be planning, see the Juniper TechLibrary for more express instructions and configurations: http://www.juniper.net/documentation/en_US/junos/topics/concept/cos-components-ex-series.html.

Recipe 2:
Migration from a Cisco LNS to vLNS Using the Subscriber Management Features on vMX

A service provider may provide a connectivity service to its own customers using the access network of another service provider. In a broadband subscriber network that access circuit is connected to the destination service provider's network using Layer 2 Tunneling Protocol (L2TP). The subscriber's point-to-point protocol (PPP) sessions are carried between the service provider networks using an L2TP tunnel from the L2TP Access Concentrator (LAC) or Broadband Network Gateway (a BNG, acting as a LAC) in the access network, to the L2TP Network Server (LNS) in the destination service provider's own network.

NOTE This recipe refrains from disclosing any details on BNG configuration or RADIUS / PPP client configuration in order to concentrate on *how to configure LNS*. It also assumes you are familiar with LNS. Look for a future *Day One* books to cover these wide-ranging topics in depth.

Problem

Since Junos OS release 15.1F6, you can configure the LNS or BNG feature set on the vMX Series. This recipe shows you how to enable the subscriber management features in Junos to configure the vMX to act as a dual stack IPv4/IPv6 virtual LNS. Since you may wish to migrate from another vendor router to vLNS, this recipe will work through an example migration from a Cisco IOS LNS to a Juniper vLNS.

Solution

With vLNS deployed on the vMX you can quickly and easily deploy an LNS into your network as shown in Figure 2.1.

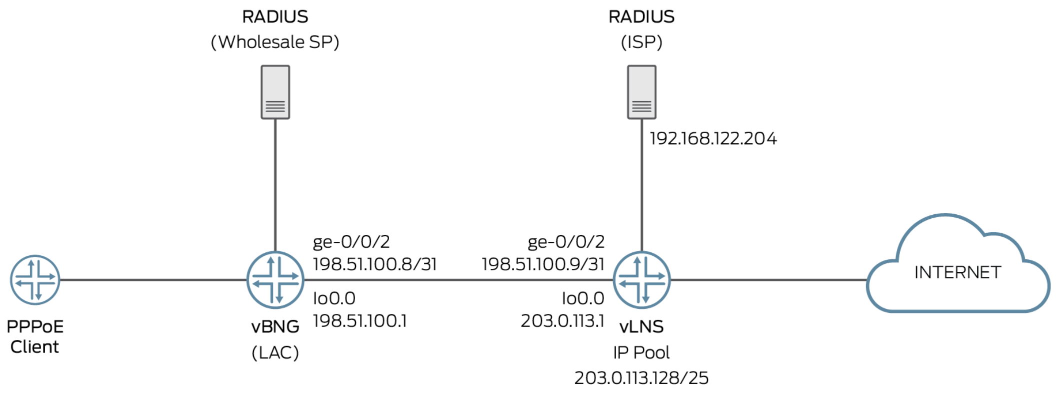


Figure 2.1 Topology Used in Configuring vLNS

While Figure 2.1's topology represents a hugely simplified broadband aggregation network, for the purposes of this recipe, the key components are present. The PPPoE subscriber is connecting directly into the LAC (in this lab, vBNG on vMX) and will be tunneled using L2TP to vLNS.

NOTE If you have not already downloaded vMX and installed a trial license key, then do so at http://www.juniper.net/us/en/dm/free-vmx-trial/ and select the key for a 60-day trial.

MORE For more information on getting going with vMX, see *Day One: vMX Up and Running* at http://www.juniper.net/us/en/training/jnbooks/day-one/automation-series/vmx-up-running/.

Okay, let's get started.

Routing Configuration

First, let's configure dynamic routing between the two SP networks. In this example the BNG and LNS are directly connected. Typically, this would not be the case in the real world but the principles for routing are the same – BGP, of course, is the protocol of choice, and on the LNS you should expect to receive the prefixes of all BNGs/LACs that will be initiating a connection to the LNS. Advertise your LNS prefixes, in this case a /32 IPv4 address assigned on vLNS for L2TP tunnel termination.

On the Cisco LNS you can see that the base interface and BGP configuration is doing what is described above:

```
interface Loopback1
 ip address 203.0.113.1 255.255.255.255
!
interface GigabitEthernet0/2
 ip address 198.51.100.9 255.255.255.254
!
router bgp 65002
 neighbor 198.51.100.8 remote-as 65001
 neighbor 198.51.100.8 description Wholesale SP
 !
 address family ipv4
  neighbor 198.51.100.8 activate
  neighbor 198.51.100.8 prefix-list AS65001-in
  neighbor 198.51.100.8 prefix-list AS65001-out
  network 203.0.113.1 255.255.255.255
 exit-address-family
 !
!
ip prefix-list AS65001-in seq 5 permit 198.51.100.1/32
ip prefix-list AS65001-out seq 5 permit 203.0.113.1/32
!
```

The Junos OS configuration follows these steps.

1. Define the interface configuration. Outside of the lab you should expect to see multiple loopback addresses defined, for management, for example, or for L2TP termination.

NOTE The interface facing the LAC/BNG should be configured with an MTU beyond the default 1500 to allow for the overhead added by the L2TP encapsulation (IP header plus UDP header plus L2TP header). Here, it is to set to 1614 (including the Layer 2 header) to allow for any future headers:

```
set interfaces lo0 unit 0 family inet address 203.0.113.1/32
set interfaces ge-0/0/2 unit 0 family inet address 198.51.100.9/31
set interfaces ge-0/0/2 mtu 1614
```

2. One difference between Cisco IOS and Junos is that on Junos the Autonomous System number is configured under the routing options stanza, whereas the rest of the BGP configuration is under protocols BGP. To configure BGP, define and configure a group "BNG" and add the neighbor to the group. You can see below the filtering of prefixes that are sent and received from the BGP neighbor: it's good practice to do this filtering outside of the lab, too:

```
set routing-options autonomous-system 65002
set protocols bgp group BNG type external
set protocols bgp group BNG import AS65001-in
set protocols bgp group BNG family inet unicast
set protocols bgp group BNG export AS65001-out
set protocols bgp group BNG peer-as 65001
set protocols bgp group BNG local-as 65002
set protocols bgp group BNG neighbor 198.51.100.8
set policy-options prefix-list AS65001-in 198.51.100.1/32
set policy-options prefix-list AS65001-out 203.0.113.1/32
set policy-options policy-statement AS65001-in term BNG from prefix-list AS65001-in
set policy-options policy-statement AS65001-in term BNG then accept
set policy-options policy-statement AS65001-in term reject then reject
set policy-options policy-statement AS65001-out term LNS from prefix-list AS65001-out
set policy-options policy-statement AS65001-out term LNS then accept
set policy-options policy-statement AS65001-out term reject then reject
```

3. Verify that the BGP session has been established and that the correct loopback prefixes are being sent and received:

```
mdinham@lns> show route advertising-protocol bgp 198.51.100.8

inet.0: 10 destinations, 10 routes (10 active, 0 holddown, 0 hidden)
  Prefix      Nexthop        MED     Lclpref    AS path
* 203.0.113.1/32          Self                                    I

mdinham@lns> show route receive-protocol bgp 198.51.100.8

inet.0: 10 destinations, 10 routes (10 active, 0 holddown, 0 hidden)
  Prefix      Nexthop        MED     Lclpref    AS path
* 198.51.100.1/32         198.51.100.8                            65001 I
```

Things look good. The loopback address is being advertised and we are receiving 198.51.100.1/32 from the LAC.

Subscriber Management Configuration

Configuration of the LNS features requires Junos enhanced subscriber management to be enabled. You must configure enhanced subscriber management to use dynamic profiles, which are an important part of the dynamic subscriber configuration on vLNS. First configure enhanced subscriber management as shown here:

1. Set the chassis mode to enhanced IP network services:

```
set chassis network-services enhanced-ip
```

2. Enable enhanced subscriber management:

```
set system services subscriber-management enable
```

3. You will need to specify a maximum size for the configuration database used by enhanced subscriber management. At the time of this writing, Juniper recommends 300MB for vMX, MX240, 480, 980, and 100MB for MX5, MX10, MX40, MX80, and MX104 routers. It's possible to further tune other configuration database parameters, for example, for virtual memory mapping, but these parameters can be left as default settings unless you have a specific reason to tune them.

```
set system configuration-database max-db-size 300M
```

4. Depending on the version of Junos that you are running, you might also need to explicitly enable the following (not required for Junos 15.1F6). The `smg-service` is the main subscriber management session management process:

```
set system processes smg-service enable
set system auto-configuration
```

5. Now commit the configuration. As you changed the network services mode to enhanced-IP and enabled subscriber management, you will be prompted to reboot the device. Take a few moments now to reboot the vMX before continuing with the configuration process:

```
mdinham@lns# commit and-quit
[edit system services subscriber-management]
  'enable'
    WARNING: Chassis configuration for subscriber-management has been changed. A system reboot
is mandatory.  Please reboot the system NOW. Continuing without a reboot might result in
unexpected system behavior.
commit complete
Exiting configuration mode

mdinham@lns> request system reboot
Reboot the system ? [yes,no] (no) yes
```

6. If you wish, when the MX has rebooted you can drop to a shell to check that the enhanced subscriber management has enabled correctly. If the parameter `net.enhanced_bbe_support` is showing as set to 1, then you are good to go:

```
mdinham@lns> start shell
% sysctl net.enhanced_bbe_support
net.enhanced_bbe_support: 1
```

The configuration of enhanced subscriber management is now complete, and it's time to get going with the actual LNS configuration.

Configuring an LNS With Inline Service Interfaces

1. To define the PPP configuration for subscribers tunneled from the LAC, configure a user group profile named `ppp-client profile`:

```
set access group-profile ppp-client-profile ppp idle-timeout 200
set access group-profile ppp-client-profile ppp ppp-options pap
set access group-profile ppp-client-profile ppp ppp-options chap
set access group-profile ppp-client-profile ppp keepalive 30
```

2. This user group profile is associated with an L2TP access profile, so all subscribers handled by a given LAC share the same PPP attributes. You can also configure PPP attributes on a per-subscriber basis if you wish (not shown here).

Here the client LAC (the vBNG in our case) is named `lac_1`. The client name should match the hostname of the LAC or the Client ID as configured on the LAC. There is also a special name "default" that represents a default tunnel client and can therefore be used as a catchall to match multiple LACs.

The client lac_1 is configured for a maximum of 2000 subscriber sessions allowed via the tunnel. Setting `interface-id` configures the ID for the interface representing an L2TP session and should match the value you configure in the `dynamic-profile dial-options` (further details below). Configuring LCP renegotiation causes the LNS to renegotiate parameters with the PPP client rather than accepting what is passed through from the LAC. The supported authentication mechanisms are specified in the referenced `user-group-profile`. This is also where you should define the shared secret tunnel password to be used between the LAC and the LNS:

```
set access profile lac1-profile client lac_1 l2tp maximum-sessions-per-tunnel 2000
set access profile lac1-profile client lac_1 l2tp interface-id lns-l2tp-id
set access profile lac1-profile client lac_1 l2tp lcp-renegotiation
set access profile lac1-profile client lac_1 l2tp shared-secret juniper
set access profile lac1-profile client lac_1 user-group-profile ppp-client-profile
```

When comparing this to your Cisco LNS, these parameters can be found by looking at the settings you have defined under `vpdn-group` and the `virtual-template`. It should be self-explanatory that you must map these settings over to your Junos configuration:

```
vpdn-group lac_1
 accept-dialin
  protocol any
  virtual-template 1
 terminate-from hostname lac_1
 lcp renegotiation on-mismatch
 l2tp tunnel password 0 juniper
!
interface Virtual-Template1
 ip unnumbered Loopback1
 peer default ip address pool p1
 ppp authentication chap radius1
 ppp authorization radius1
 ppp accounting radius1
!
```

3. Now configure an AAA access-profile as this will enable you to override the global AAA configuration and apply the AAA profile to a group of tunnels. It's in this profile that you define all the important details for your RADIUS server, such as IP addressing, ports, and secret key. If you have a preferred source address, set it here also:

```
set access profile radius1 authentication-order radius
set access profile radius1 radius authentication-server 192.168.122.204
set access profile radius1 radius accounting-server 192.168.122.204
set access profile radius1 radius-server 192.168.122.204 port 1812
set access profile radius1 radius-server 192.168.122.204 accounting-port 1813
set access profile radius1 radius-server 192.168.122.204 secret juniper
set access profile radius1 radius-server 192.168.122.204 timeout 10
set access profile radius1 radius-server 192.168.122.204 retry 4
set access profile radius1 radius-server 192.168.122.204 source-address 203.0.113.1
set access profile radius1 accounting order radius
```

To define multiple RADIUS servers configure another radius server and specify the IP address of the additional servers at set access profile radius1 radius authentication-server and at set access profile radius1 radius accounting-server.

Let's look again at the Cisco LNS configuration. The configuration is different but easy to compare with, and move to, Junos. The group radius1 is specified in the virtual-template interface settings:

```
aaa authentication ppp radius1 group radius
aaa authorization network radius1 group radius
aaa accounting network radius1 start-stop group radius
ip radius source-interface Loopback0
radius-server host 192.168.122.204 auth-port 1812 acct-port 1813
radius-server timeout 10
radius-server key juniper
```

4. In this example, our PPP subscriber will be assigned an IPv4 address via a local address pool. On Junos it's executed as follows and the pool is named p1. If you like, you can also set the upper and lower boundaries of the addresses in the pool range. A domain map named default matches on all subscriber domains (realms) and specifies the `address-pool` to be used by all subscribers matching the domain map. If you prefer, specify a non-default realm (the pool could also be assigned to a subscriber via RADIUS):

```
set access address-assignment pool p1 family inet network 203.0.113.128/25
set access address-assignment pool p1 family inet range p1-range low 203.0.113.128
set access address-assignment pool p1 family inet range p1-range high 203.0.113.191
set access domain map default address-pool p1
```

NOTE Remember to advertise the subscriber pool into your own network's routing.

When comparing this to the Cisco configuration, your keen eyes might have spotted the pool being referenced in the virtual-template shown above. The pool itself would be defined like this: ip local pool p1 203.0.113.128 203.0.113.255.

5. Now configure the inline service interface. This is a virtual anchor interface on the packet forwarding engine and allows you to use L2TP services without a services PIC. On the MX Series, this requires an MPC. On the vMX you configure si-0/0/0 but this interface will vary for other MX platforms. Choose a suitable bandwidth as required for your implementation:

```
set chassis fpc 0 pic 0 inline-services bandwidth 1g
```

The anchor interface itself must then be configured and assigned to a service device pool. This pool will be referenced in an L2TP tunnel group, which we will configure next.

```
set interfaces si-0/0/0 encapsulation generic-services
set interfaces si-0/0/0 unit 0 family inet
set services service-device-pools pool lns_p1 interface si-0/0/0
```

6. Now define the L2TP tunnel group (here named `tg-dynamic`) that specifies attributes applicable to the L2TP tunnels and sessions from a group of LAC clients. The tunnel group references the access profile `lac1-profile` that was defined above and validates connections to the LNS:

```
set services l2tp tunnel-group tg-dynamic l2tp-access-profile lac1-profile
```

Specify the AAA profile to authenticate subscribers. This setting overrides the global setting, if it is defined (e.g. `set access-profile radius1`):

```
set services l2tp tunnel-group tg-dynamic aaa-access-profile radius1
```

The local gateway is the interface address on the LNS that terminates tunnels from the LAC, usually a loopback interface address.

```
set services l2tp tunnel-group tg-dynamic local-gateway address 203.0.113.1
```

Specify the pool of inline service anchor interfaces that was defined earlier:

```
set services l2tp tunnel-group tg-dynamic service-device-pool lns_p1
```

And finally, the name of the dynamic profile that defines and configures the per-subscriber inline service interfaces for L2TP sessions:

```
set services l2tp tunnel-group tg-dynamic dynamic-profile lns-client-profile
```

7. This example uses a dynamic profile named `lns-client-profile`. Dynamic profiles can look a bit confusing at first glance but we are setting the parameters for the subscriber sessions that tunnel to the LNS, kind of like the virtual template interface on a Cisco LNS. The variables (beginning with `$`) are dynamically set by the Junos OS when the per-subscriber interface is created.

The logical subscriber interfaces are defined by the anchor service interface (`$junos-interface-ifd-name`) and the logical unit number (`$junos-interface-unit`). Now create the dynamic profile and use `lns-l2tp-id` as the `interface-id` to match what was set earlier in the LAC client access profile. The interface is set to `dedicated` to indicate that the logical interface can only relate to one subscriber at a time:

```
set dynamic-profiles lns-client-profile interfaces "$junos-interface-ifd-name" unit "$junos-
interface-unit" dial-options l2tp-interface-id lns-l2tp-id
set dynamic-profiles lns-client-profile interfaces "$junos-interface-ifd-name" unit "$junos-
interface-unit" dial-options dedicated
```

To enable IPv4 for the subscriber, configure the inet address family for the logical interfaces and enable the IP address of the loopback interface that provides the local termination for the L2TP tunnel:

```
set dynamic-profiles lns-client-profile interfaces “$junos-interface-ifd-
name” unit “$junos-interface-unit” family inet unnumbered-address “$junos-loopback-
interface”
```

Now, if you would like to route additional prefixes to the subscriber there is a small amount of additional configuration to the dynamic profile by configuring the routing options for the access routes. The configuration is applied in a similar way to what you have already been doing.

Access and Access-Internal Routes

DHCP and PPP on Junos represent a subscriber as a /32 access-internal route and the networks routed to the subscriber as access routes. Access routes are generally used to apply the Framed-Route (Framed-IPv4-Route) and (Framed-IPv6-Route) attributes from RADIUS. A Framed-Route only needs to specify the route prefix, but optionally can include next-hop, metric, preference, and tag. If the next hop is not explicitly specified, then its absence implies a next-hop of 0.0.0.0, which must resolve to the subscriber CPE IP address.

Here the PPP subscriber interface (`$junos-interface-name`) is dynamically created in the routing instance (`$junos-routing-instance`) that's assigned to the subscriber:

```
set dynamic-profiles lns-client-profile routing-instances “$junos-routing-
instance” interface “$junos-interface-name”
```

Routing options for access routes are the usual options and include next-hop address (`$junos-framed-route-nexthop`), metric (`$junos-framed-route-cost`), and preference (`$junos-framed-route-distance`).

The next configuration creates the per subscriber routes that have been passed to the LNS from the RADIUS server as access routes:

```
set dynamic-profiles lns-client-profile routing-instances “$junos-routing-
instance” routing-options access route $junos-framed-route-ip-address-prefix next-
hop “$junos-framed-route-nexthop”
set dynamic-profiles lns-client-profile routing-instances “$junos-routing-
instance” routing-options access route $junos-framed-route-ip-address-
prefix metric “$junos-framed-route-cost”
set dynamic-profiles lns-client-profile routing-instances “$junos-routing-
instance” routing-options access route $junos-framed-route-ip-address-
prefix preference “$junos-framed-route-distance”
set dynamic-profiles lns-client-profile routing-instances “$junos-routing-
instance” routing-options access route $junos-framed-route-ip-address-prefix tag “$junos-
framed-route-tag”
```

NOTE Starting in release 15.1R1, you no longer need to include the access-internal stanza in the dynamic-profile when the `access` stanza is present for Framed-Route support. If the RADIUS Framed-Route attribute does not specify the next-hop, then the variable `$junos-framed-route-nexthop` defaults to 0.0.0.0. As this implies, the subscriber address should be used; the next hop is automatically resolved by Junos using the subscriber IP address stored in the session database.

If Framed-Routes are not being used, a dynamic IP address variable (`$junos-subscriber-ip-address`) configures the routing options for access-internal routes in the routing instance:

```
set dynamic-profiles lns-client-profile routing-instances "$junos-routing-
instance" routing-options access-internal route $junos-subscriber-ip-address qualified-
next-hop "$junos-interface-name"
```

Now would be a good time to adjust your RE filters, if you've not done so already. You will need to permit UDP port 1701 between the LAC and the LNS to allow the traffic. An example policy is shown below, which references the prefix list created earlier. Before applying this policy to the loopback interface, add terms as needed to allow any other traffic that should also be permitted:

```
set firewall family inet filter RE-FILTER term accept-l2tp from source-prefix-
list AS65001-in
set firewall family inet filter RE-FILTER term accept-l2tp from protocol udp
set firewall family inet filter RE-FILTER term accept-l2tp from destination-port 1701
set firewall family inet filter RE-FILTER term accept-l2tp then count accept-l2tp
set firewall family inet filter RE-FILTER term accept-l2tp then accept
set firewall family inet filter RE-FILTER term deny-l2tp from protocol udp
set firewall family inet filter RE-FILTER term deny-l2tp from destination-port 1701
set firewall family inet filter RE-FILTER term deny-l2tp then count reject-l2tp
set firewall family inet filter RE-FILTER term deny-l2tp then reject
```

And that's it – the LNS is configured, now to fire up your PPP client and test the connection.

NOTE If you are doing this on the vMX, even in the lab, be sure to have the trial license applied before you attempt a subscriber connection, otherwise you won't see the sessions establish.

vBNG/vLNS Licensing

At the time of this writing, using the vMX in a live environment as a virtual broadband network gateway (BNG) or virtual L2TP network server (vLNS) requires a vMX Premium bandwidth license for each virtual BNG/LNS instance and a Broadband Subscriber Scale license. Broadband Subscriber Scale licenses are available in three packages: *Introductory*, *Preferred*, and Elite.

Introductory Package – L2TP feature set (LNS), Secure Policy/Lawful Intercept, Service Activation/Deactivation via RADIUS, RE-based HTTP Redirect, e.g. LNS wholesale, Walled garden (billing).

Preferred Package – All Introductory features plus DHCP subscriber services, PPP/LAC subscriber services, DHCP Relay and Local Server, e.g. Residential BBE, Broadband business services, L2TP LAC.

Elite Package – All Preferred features, plus Wireless Policy Management via Gx (PCEF), Wireline online charging via Gy (PCEF), Pseudowire Headend Termination, Advanced Multicast Video QoS.

MORE? For more details see the vMX Datasheet available at http://www.juniper.net/assets/us/en/local/pdf/datasheets/1000522-en.pdf.

Subscriber IP addressing

There are two ways to allocate IP addresses to your subscribers, either as a local pool of addressing, as shown earlier, or directly from RADIUS. Let's look at the options.

By using a domain map you will be able to configure a map that specifies various options for all subscribers in a domain/realm. This map can be applied to a specific domain/realm or as a default applied to all users. The default is created like this:

```
set access domain map default address-pool p1
```

This simply says use address pool `p1` for all subscribers. Of course, this could be overridden via RADIUS, or if you wanted to apply a domain map to a specific domain/realm:

```
set access domain map example.com address-pool p2
```

Now, let's look at how to allocate addressing to a subscriber in RADIUS. It's all done via the profile for the subscriber. To configure the subscriber session to use an IP address from a local pool, use the RADIUS attribute "Framed-Pool" specific to the LNS pool. For example, to specify using a pool named "p2" the RADIUS profile would look like this:

```
juniper@example.net   Cleartext-Password := "juniper"
                      Framed-Pool = "p2"
```

Alternatively, you can use RADIUS to assign a specific IP address to the subscriber and that's done using the attributes "Framed-IP-Address" and "Framed-IP-Netmask", and you can also route a prefix at the subscriber:

```
  juniper@example.net   Cleartext-Password := "juniper"
Framed-IP-Address = 208.203.0.129,
Framed-IP-Netmask = 255.255.255.255,
Framed-Route = "208.203.0.64/28 208.203.0.129 1"
```

Verification

Now to validate and troubleshoot your configuration. Assuming RADIUS is working correctly and you have a PPP client attempting to connect, then you will see the subscriber session on the LNS. To check that the subscriber has connected, you can do the following:

```
mdinham@lns> show subscribers
Interface           IP Address/VLAN ID                User Name                     LS:RI
si-0/0/0.3221225498 208.203.0.129                     juniper@example.
net       default:default
```

You can see the virtual interface and notice that it is a sub-interface of the service interface that was defined earlier. It's also possible to use the `extensive` option to find out more details on the subscriber:

```
mdinham@lns> show subscribers extensive
Type: L2TP
User Name: juniper@example.net
IP Address: 208.203.0.129
IP Netmask: 255.255.255.255
Logical System: default
Routing Instance: default
Interface: si-0/0/0.3221225498
Interface type: Dynamic
Underlying Interface: si-0/0/0.3221225498
Dynamic Profile Name: lns-client-profile
State: Active
Radius Accounting ID: 27
Session ID: 27
PFE Flow ID: 39
Login Time: 2017-01-17 23:29:31 UTC
Accounting interval: 0
Dynamic configuration:
  junos-framed-route-ip-address-prefix: 208.203.0.64/28
      junos-framed-route-nexthop: 208.203.0.129
          junos-framed-route-cost: 1
```

Notice that you can now see details of the prefix that was routed to the subscriber as well as other useful information such as the dynamic profile that is in use and the subscriber login time.

You can also look up details on the subscriber like this, specify the Session ID to see more information:

```
mdinham@lns> show network-access aaa subscribers
Username                 Logical system/Routing instance   Client type    Session-ID
juniper@example.net      default:default                   l2tp           27

mdinham@lns> show network-access aaa subscribers session-id 27
Logical system/Routing instance   Client type    Session-
```

```
ID     Session uptime    Accounting
default:default                   l2tp          27             00:06:29         on/time
```

Well, none of this would be possible without L2TP, so here's how to look at the status of the L2TP tunnels between the LAC/BNG and the LNS. There's only one tunnel here, but in a live network, you should expect to see many:

```
mdinham@lns> show services l2tp session
Tunnel local ID: 20080
  Local  Remote   State                Interface         Interface
  ID     ID                            unit              Name
  33419  52912    Established          3221225498        si-0/0/0

mdinham@lns> show services l2tp tunnel
  Local ID  Remote ID  Remote IP               Sessions  State
  20080     11032      198.51.100.1:1701              1  Established
```

Again, specifying the `extensive` option gives you much more information on the L2TP tunnel:

```
mdinham@lns> show services l2tp session extensive
Tunnel local ID: 20080
  Session local ID: 33419, Session remote ID: 52912
    Interface unit: 3221225498
    State: Established
    Interface: si-0/0/0
    Mode: Dedicated
    Local IP: 203.0.113.1:1701, Remote IP: 198.51.100.1:1701
    Local name: lns, Remote name: lac_1
    Bearer type: 1, Framing type: 1
    LCP renegotiation: On, Authentication: None, Interface ID: si-0/0/0
    Call serial number: 120
    Tx speed: 1000000000, Rx speed: 0
    Create time: Tue Jan 17 23:29:31 2017, Up time: 00:09:16
    Idle time: N/A, ToS Reflect: Disabled
    Statistics since: Tue Jan 17 23:29:31 2017
                        Packets       Bytes
      Data Tx                 0           0
      Data Rx                 0           0

mdinham@lns> show services l2tp tunnel extensive
Waiting for statistics...
  Tunnel local ID: 20080, Tunnel remote ID: 11032
    Remote IP: 198.51.100.1:1701
    Sessions: 1, State: Established, Administrative State: Enabled
    Tunnel Name: 2/12
    Local IP: 203.0.113.1:1701
    Local name: lns, Remote name: lac_1
    Effective Peer Resync Mechanism: failover protocol
    Nas Port Method: none
    Tunnel Logical System: default, Tunnel Routing Instance: default
    Max sessions: 2000, Window size: 4, Hello interval: 60
    Create time: Tue Jan 17 23:28:57 2017, Up time: 00:09:43
    Idle time: 00:00:00, ToS Reflect: Disabled
    Interface Name: lns_p1, Tunnel Group Name: tg-dynamic
    Statistics since: Tue Jan 17 23:28:57 2017
                        Packets       Bytes
      Control Tx             16         439
      Control Rx            18          904
      Data Tx                0            0
      Data Rx                0            0
      Errors Tx              0
      Errors Rx              0
```

Troubleshooting

If you are finding that subscribers are not connecting it can be useful to turn on some traceoptions for debugging. The general-authentication-service logs will show and authentication or RADIUS related issues:

```
set system processes general-authentication-service traceoptions file authlog
set system processes general-authentication-service traceoptions flag all
```

You should use specific traceoptions flags outside of the lab because "flag all" will generate a lot of data. For example, if a subscriber is using an incorrect password you will see the following in the logs:

```
Feb 13 17:57:10.424160 RADIUS server 192.168.122.204:1812 was used for last request
Feb 13 17:57:10.424174 authd_radius_callback: RADIUS server sent an ACCESS_
REJECT, failing login for session-id:8
Feb 13 17:57:10.424187 loadDefaultService:: default service for the subscriber is empty
Feb 13 17:57:10.424191 Radius result is CLIENT_REQ_STATUS_SUCCESS
Feb 13 17:57:10.424225 Parsing RADIUS message for session-id:8
Feb 13 17:57:10.424252 Framework - module(radius) return: FAILURE
```

For PPP issues, it's useful to trace the PPP service:

```
set protocols ppp-service traceoptions file ppp-service
set protocols ppp-service traceoptions flag all
```

IPv6 Subscribers

But what about IPv6? Configuring IPv6 on LNS is not too different from what has already been configured, but it does require a few additional steps. Let's get going.

NOTE There are several ways that an ISP could assign prefixes to an IPv6 subscriber. This recipe demonstrates a common method that dynamically allocates the subscriber WAN address using NDRA and the knowledge of IPv6 address as subscriber LAN address using DHCP-PD. This recipe assumes some knowledge of IPv6 address assignment. Look for a future *Day One* book to cover this topic in depth.

The recipe assumes that a base-level IPv6 configuration is already applied, for example, loopback addressing and onward network routing:

```
set interfaces lo0 unit 0 family inet6 address 2001:db8:0:1::1/128
```

There isn't too much additional configuration required on the LNS to enable dual stack IPv4/IPv6. Now configure the routing options for access routes. The next configuration creates the per-subscriber routes that have been passed to the LNS from the RADIUS server. Notice that although the variables have changed, the configuration is very similar to what you configured earlier for IPv4 routes; except for IPv6 a RIB is specified:

```
set dynamic-profiles lns-client-profile routing-instances "$junos-routing-
instance" routing-options rib "$junos-ipv6-rib" access route $junos-framed-route-ipv6-
address-prefix next-hop "$junos-framed-route-ipv6-nexthop"

set dynamic-profiles lns-client-profile routing-instances "$junos-routing-
instance" routing-options rib "$junos-ipv6-rib" access route $junos-framed-route-ipv6-
address-prefix qualified-next-hop "$junos-interface-name"

set dynamic-profiles lns-client-profile routing-instances "$junos-routing-
instance" routing-options rib "$junos-ipv6-rib" access route $junos-framed-route-ipv6-
address-prefix metric "$junos-framed-route-ipv6-cost"

set dynamic-profiles lns-client-profile routing-instances "$junos-routing-
instance" routing-options rib "$junos-ipv6-rib" access route $junos-framed-route-ipv6-
address-prefix preference "$junos-framed-route-ipv6-distance"

set dynamic-profiles lns-client-profile routing-instances "$junos-routing-
instance" routing-options rib "$junos-ipv6-rib" access route $junos-framed-route-ipv6-
address-prefix tag "$junos-framed-route-ipv6-tag"
```

NOTE If you try to amend a dynamic-profile whilst a subscriber is connected on the same profile, Junos will not allow you to commit the configuration. You can configure `set system dynamic-profile-options versioning` and this will allow the amendment to be committed. Subscribers will pick up the changed profile as they reconnect.

As mentioned earlier, from 15.1R1 the `access-internal` stanza is no longer required in the dynamic profile when the `access` stanza is present for framed route support:

```
set dynamic-profiles lns-client-profile routing-instances "$junos-routing-
instance" routing-options rib "$junos-ipv6-rib" access-internal route $junos-subscriber-
ipv6-address qualified-next-hop "$junos-interface-name"
```

Configure the address family for the logical subscriber interfaces and enable the local address on the LNS that provides termination for the L2TP tunnel as an unnumbered address derived from the loopback interface:

```
set dynamic-profiles lns-client-profile interfaces "$junos-interface-ifd-
name" unit "$junos-interface-unit" family inet6 unnumbered-address $junos-loopback-
interface
```

Since this will be accomplished using NDRA, the configuration is slightly different – a numbered address is configured rather than an unnumbered address for the IPv6 local address. The variable `$junos-`

`ipv6-address` represents the IPv6 address of the interface used for router advertisements:

```
set dynamic-profiles lns-client-profile interfaces "$junos-interface-ifd-
name" unit "$junos-interface-unit" family inet6 address $junos-ipv6-address
```

Next, enable NDRA to assign an IPv6 address on the WAN link. This prefix could be assigned locally from a pool or assigned via RADIUS:

```
set dynamic-profiles lns-client-profile protocols router-advertisement interface "$junos-
interface-name" prefix $junos-ipv6-ndra-prefix
```

To use DHCPv6 Prefix Delegation (DHCP-PD) to assign an IPv6 for subscriber LAN addressing, the Junos local DHCP server is used. This prefix could be assigned locally from a pool or assigned via RADIUS. The interface is the inline services anchor interface that was configured earlier, here si-0/0/0.0:

```
set system services dhcp-local-server dhcpv6 group v6-ppp-subscriber interface si-0/0/0.0
```

In this recipe, the subscriber WAN address and LAN address will be allocated dynamically from local pools on the LNS. Now let's create two address assignment pools, one for NDRA and one for DHCP-PD.

Here a pool named `ndra-pool` is used for assigning the WAN address from a pool of size /48. Each subscriber will be allocated a /64:

```
set access address-assignment pool ndra-pool family inet6 prefix 2001:db8:1000::/48
set access address-assignment pool ndra-pool family inet6 range r1 prefix-length 64
```

Here a pool named `dhcp-pd-pool` is used for assigning the LAN address from a pool of size /36. Each subscriber will be allocated a /48:

```
set access address-assignment pool dhcp-pd-pool family inet6 prefix 2001:db8:2000::/36
set access address-assignment pool dhcp-pd-pool family inet6 range r1 prefix-length 48
```

Now that the pools are created you must tell Junos to use them. This can be done in RADIUS by using the attributes Framed-IPv6-Pool to specify the NDRA pool, and the attribute Delegated-IPv6-Prefix-Pool to specify the DHCPv6 PD pool. Rather than doing it via RADIUS, let's configure Junos to use the pools just created by default.

You specify the default NDRA pool like this:

```
set access address-assignment neighbor-discovery-router-advertisement ndra-pool
```

And the DHCPv6 pool like this:

```
set system services dhcp-local-server dhcpv6 overrides delegated-pool dhcp-pd-pool
```

NOTE If using RADIUS to assign the WAN IPv6 Prefix or Pool, it can be useful to use the Framed-Interface-Id attribute as this allows you to control the IPv6 address that the subscriber will configure. For example, using a Framed-Interface-Id of "0:0:0:2" will set the last 64 bits of the subscriber address to be 0:0:0:2.

IPv6 Verification

Okay, the LNS is configured, so now it's time for some verification. This time when you look at the connected subscribers, the IPv6 prefixes are shown too:

```
mdinham@lns> show subscribers
Interface            IP Address/VLAN ID          User Name                     LS:RI
si-0/0/0.3221225558  208.203.0.129               juniper@example.net       default:default
*                    2001:db8:1000:2::/64
si-0/0/0.3221225558  2001:db8:2011::/48                                    default:default
```

Notice that if the subscriber disconnects and reconnects that a different set of IPv6 prefixes will be assigned from the pool:

```
mdinham@lns> show subscribers
Interface            IP Address/VLAN ID          User Name                     LS:RI
si-0/0/0.3221225560  208.203.0.129               juniper@example.net       default:default
*                    2001:db8:1000:3::/64
si-0/0/0.3221225560  2001:db8:2012::/48                                    default:default
```

Adding the `detail` option will provide more information on how the IPv6 prefix has been assigned:

```
mdinham@lns> show subscribers detail
Type: L2TP
User Name: juniper@example.net
IP Address: 208.203.0.129
IP Netmask: 255.255.255.255
IPv6 User Prefix: 2001:db8:1000:3::/64
Logical System: default
Routing Instance: default
Interface: si-0/0/0.3221225560
Interface type: Dynamic
Underlying Interface: si-0/0/0.3221225560
Dynamic Profile Name: lns-client-profile
State: Active
Radius Accounting ID: 101
Session ID: 101
PFE Flow ID: 101
Login Time: 2017-01-30 06:26:06 UTC

Type: DHCP
IPv6 Prefix: 2001:db8:2012::/48
Logical System: default
Routing Instance: default
Interface: si-0/0/0.3221225560
Interface type: Static
Underlying Interface: si-0/0/0.3221225560
MAC Address: 52:54:00:66:b5:57
State: Active
Radius Accounting ID: jnpr :102
Session ID: 102
Underlying Session ID: 101
PFE Flow ID: 101
Login Time: 2017-01-30 06:26:08 UTC
DHCP Options: len 48
00 01 00 0e 00 01 00 01 20 20 b4 6c 52 54 00 66 b5 57 00 08
```

```
00 02 00 00 00 06 00 04 00 17 00 18 00 19 00 0c 00 00 00 01
00 00 00 00 00 00 00 00
```

To verify the pool used for NDRA and the delegated pool that is used for DHCPv6 prefix delegation, use the `extensive` option to show the length of prefixes that were delegated and the IPv6 address that is configured on the subscriber interface on the LNS:

```
mdinham@lns> show subscribers extensive | match IPv6
IPv6 User Prefix: 2001:db8:1000:3::/64
IPv6 Address Pool: ndra-pool
IPv6 Interface Address: 2001:db8:1000:3::1/64
IPv6 Framed Interface Id: 4baa:89a0:8fa8:f572
  junos-ipv6-ndra-prefix: 2001:db8:1000:3::/64
IPv6 Prefix: 2001:db8:2012::/48
IPv6 Address Pool: ndra-pool
IPv6 Delegated Address Pool: dhcp-pd-pool
```

You can also see the IPv6 address bindings on the local DHCP server. The prefix field shows DHCPv6 prefix assigned to the subscriber:

```
mdinham@lns> show dhcpv6 server binding
Prefix                  Session Id  Expires  State    Interface    Client DUID
2001:db8:2012::/48      102         85875    BOUND    si-0/0/0.3221225560 LL_TIME0x1-
0x2020b46c-52:54:00:66:b5:57
```

Discussion

Using this recipe, you will be able to deploy a Juniper LNS on a MX Series or a vMX. The configuration might look a bit different than the Cisco LNS at first glace, but once everything is broken down into components there's no problem migrating the configuration. There's a lot of flexibility in the way an LNS can be configured on the Junos OS, particularly when the LNS is terminating L2TP tunnels from multiple service providers.

Recipe 3:
Achieving Multi-Path in Route Reflection Using BGP Add Path

This cookbook recipe is similar to Recipe 11, *Migrate Your Core to Centralized Route Reflection and Segment Routing*, in that we're utilizing BGP add-path. But here you will look at the issue of using route reflection within a given routing environment, which dramatically assists with the scaling of iBGP as it eliminates the full-mesh iBGP requirement. While route reflection (RR) can simplify and help a network to scale, as discussed in Recipe 11, this recipe considers an issue when using the default parameters that can minimize the ability to perform equal-cost-multipathing (ECMP) in an environment. There's nothing like a new knob in Junos to attract attention. Let's jump into the advantages of BGP multipathing despite the fact that we're using a route reflector.

Problem

Under normal circumstances, in a network using traditional iBGP peering, if an identical route is introduced via multiple peers, a peer receiving such routes will see that it has multiple paths to reach a given destination. Let's take a look at this behavior under normal circumstances to see how routing behaves in a traditional iBGP peering relationship, using the topology illustrated in Figure 3.1

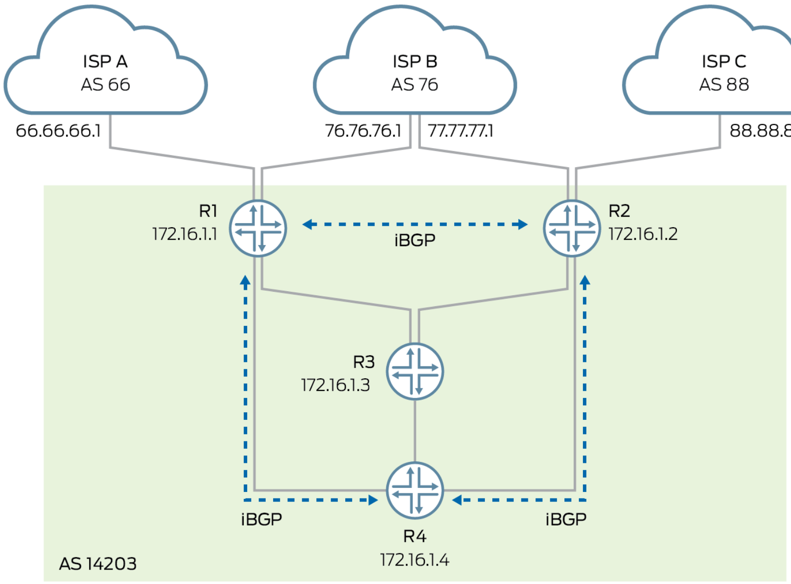


Figure 3.1 Traditional iBGP Peering

You can see in Figure 3.1 that Autonomous System 14023 is comprised of four iBGP neighbors. In this case R1 and R2 have peering relationships to the outside world via eBGP, and they also peer internally to each other and also to R4. Peering is done from loopback to loopback, and the IGP in this case is OSPF. Both R1 and R2 have a peering relationship to ISP B with an Autonomous System number of 76. In this case, identical routes are being advertised from ISP B to both R1 and R2.

Let's take a look at R4 to see how it perceives routes that are ingested from R1 and R2:

```
root@r4> show bgp summary
Groups: 1 Peers: 2 Down peers: 0
Table          Tot Paths  Act Paths Suppressed    History Damp State    Pending
inet.0                 8          6          0          0          0          0
Peer                     AS      InPkt     OutPkt    OutQ   Flaps Last Up/
Dwn State|#Active/Received/Accepted/Damped...
172.16.1.1          14023          6          5       0       0      1:25 4/4/4/0      0/0/0/0
172.16.1.2          14023         13         12       0       1       4:08 2/4/4/0      0/0/0/0

root@r4> show route protocol bgp

inet.0: 21 destinations, 23 routes (21 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both
```

```
66.66.66.0/24      *[BGP/170] 00:01:33, localpref 100, from 172.16.1.1
                      AS path: 66 I
                    > to 10.0.1.9 via ge-0/0/1.0
67.67.67.0/24      *[BGP/170] 00:01:33, localpref 100, from 172.16.1.1
                      AS path: 66 I
                    > to 10.0.1.9 via ge-0/0/1.0
76.76.76.0/24      *[BGP/170] 00:01:33, localpref 100, from 172.16.1.1
                      AS path: 76 I
                    > to 10.0.1.9 via ge-0/0/1.0
                    [BGP/170] 00:04:16, localpref 100, from 172.16.1.2
                      AS path: 76 I
                    > to 10.0.1.13 via ge-0/0/2.0
77.77.77.0/24      *[BGP/170] 00:01:33, localpref 100, from 172.16.1.1
                      AS path: 76 I
                    > to 10.0.1.9 via ge-0/0/1.0
                    [BGP/170] 00:04:16, localpref 100, from 172.16.1.2
                      AS path: 76 I
                    > to 10.0.1.13 via ge-0/0/2.0
88.88.88.0/24      *[BGP/170] 00:04:16, localpref 100, from 172.16.1.2
                      AS path: 88 I
                    > to 10.0.1.13 via ge-0/0/2.0
89.89.89.0/24      *[BGP/170] 00:04:16, localpref 100, from 172.16.1.2
                      AS path: 88 I
                    > to 10.0.1.13 via ge-0/0/2.0
```

In this case, you can see that each upstream autonomous system is advertising two routes, and in the case of routes being advertised via ISP A and ISPC, there is only a single next hop because these upstream ISPs are only single-homed into AS 14023. However, in the case of the 76.76.76.0/24 and 77.77.77.0/24 routes advertised by ISPB, you'll see a total of two next hops on R4 because ISPB is multi-homed to R1 and R2. Let's get more details about this at 76.76.76.0/24:

```
root@r4> show route 76.76.76.0/24 extensive

inet.0: 21 destinations, 23 routes (21 active, 0 holddown, 0 hidden)
76.76.76.0/24 (2 entries, 1 announced)
TSI:
KRT in-kernel 76.76.76.0/24 -> {indirect(262143)}
        *BGP    Preference: 170/-101
                Next hop type: Indirect
                Address: 0x934cc28
                Next-hop reference count: 12
                Source: 172.16.1.1
                Next hop type: Router, Next hop index: 568
                Next hop: 10.0.1.9 via ge-0/0/1.0, selected
                Protocol next hop: 172.16.1.1
                Indirect next hop: 942c910 262143
                State: <Active Int Ext>
                Local AS: 14023 Peer AS: 14023
                Age: 1:41      Metric2: 1
                Task: BGP_14023.172.16.1.1+179
```

```
              Announcement bits (2): 0-KRT 4-Resolve tree 1
              AS path: 76 I
              Accepted
              Localpref: 100
              Router ID: 172.16.1.1
              Indirect next hops: 1
                      Protocol next hop: 172.16.1.1 Metric: 1
                      Indirect next hop: 942c910 262143
                      Indirect path forwarding next hops: 1
                              Next hop type: Router
                              Next hop: 10.0.1.9 via ge-0/0/1.0
                      172.16.1.1/32 Originating RIB: inet.0
                        Metric: 1                       Node path count: 1
                        Forwarding nexthops: 1
                              Nexthop: 10.0.1.9 via ge-0/0/1.0
       BGP    Preference: 170/-101
              Next hop type: Indirect
              Address: 0x934ca30
              Next-hop reference count: 8
              Source: 172.16.1.2
              Next hop type: Router, Next hop index: 569
              Next hop: 10.0.1.13 via ge-0/0/2.0, selected
              Protocol next hop: 172.16.1.2
              Indirect next hop: 942c828 262142
              State: <NotBest Int Ext>
              Inactive reason: Not Best in its group - Router ID
              Local AS: 14023 Peer AS: 14023
              Age: 4:24       Metric2: 1
              Task: BGP_14023.172.16.1.2+52339
              AS path: 76 I
              Accepted
              Localpref: 100
              Router ID: 172.16.1.2
              Indirect next hops: 1
                      Protocol next hop: 172.16.1.2 Metric: 1
                      Indirect next hop: 942c828 262142
                      Indirect path forwarding next hops: 1
                              Next hop type: Router
                              Next hop: 10.0.1.13 via ge-0/0/2.0
                      172.16.1.2/32 Originating RIB: inet.0
                        Metric: 1                       Node path count: 1
                        Forwarding nexthops: 1
                              Nexthop: 10.0.1.13 via ge-0/0/2.0
```

And you can see that there are two next hops in the routing table, although one of them is inactive due to the router ID selection in the BGP path selection algorithm.

NOTE This is to be expected due to the fact that multipathing is not enabled in any way. The important thing to stress here is that you are in fact receiving identical routes from both R1 and R2 for the routes advertised by ISPB, and R4 can make a local decision as to which one of these is preferred. You can then go on to enable multipathing and add a forwarding table load-balancing policy for true multipathing.

Now let's compare and contrast this behavior to one in which route reflection is used in the network, as shown in Figure 3.2. In this case, R3 is the RR, and the direct peering relationships between all of our devices has been removed. So all devices will simply perform iBGP peering with R3, as shown in Figure 3.2.

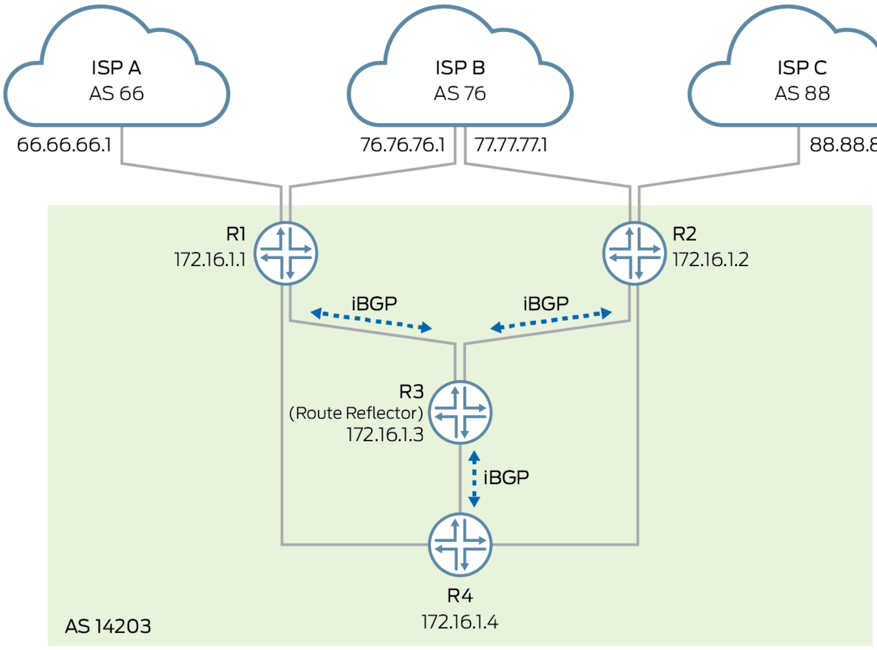


Figure 3.2 iBGP Route Reflection

Before diving too deeply into the output, let's take a look at the configuration at both R3 (the route reflector) and R4 (the route reflector client):

```
root@r4> **show route 76.76.76.0/24 extensive**

R3:
[edit protocols bgp]
root@r3# show
group RR_Clients {
    type internal;
    local-address 172.16.1.3;
    cluster 1.1.1.1;
    neighbor 172.16.1.1;
    neighbor 172.16.1.2;
    neighbor 172.16.1.4;
}


R4:
[edit protocols bgp]
root@r4# show
group ibgp {
    type internal;
    local-address 172.16.1.4;
    neighbor 172.16.1.3;
}
```

As you can see, there's a normal peering relationship where R3 is the route reflector using a cluster ID of 1.1.1.1 and from the perspective of R4, R3 simply sees it has a traditional iBGP peering relationship with R4. Next you can also see that R3 is receiving routes from both R1 and R2, and has multiple potential next hops available for the multi-homed routes from ISP B:

```
root@r3> show bgp summary
Groups: 1 Peers: 3 Down peers: 0
Table         Tot Paths  Act Paths Suppressed    History Damp State    Pending
inet.0              8        6         0         0         0         0
Peer                  AS     InPkt     OutPkt    OutQ   Flaps Last Up/Dwn State|#Active/
Received/Accepted/Damped...
172.16.1.1          14023        8         9      0      2       1:53 4/4/4/0
0/0/0/0
172.16.1.2          14023       51        56      0      1      12:43 2/4/4/0
0/0/0/0
172.16.1.4          14023       31        39      0      1      12:43 0/0/0/0
0/0/0/0

root@r3> show route protocol bgp

inet.0: 21 destinations, 23 routes (21 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

66.66.66.0/24      *[BGP/170] 00:01:56, localpref 100, from 172.16.1.1
                      AS path: 66 I
                    > to 10.0.1.1 via ge-0/0/1.0
67.67.67.0/24      *[BGP/170] 00:01:56, localpref 100, from 172.16.1.1
                      AS path: 66 I
                    > to 10.0.1.1 via ge-0/0/1.0
76.76.76.0/24      *[BGP/170] 00:01:56, localpref 100, from 172.16.1.1
                      AS path: 76 I
                    > to 10.0.1.1 via ge-0/0/1.0
                    [BGP/170] 00:02:34, localpref 100, from 172.16.1.2
                      AS path: 76 I
                    > to 10.0.1.5 via ge-0/0/2.0
77.77.77.0/24      *[BGP/170] 00:01:56, localpref 100, from 172.16.1.1
                      AS path: 76 I
                    > to 10.0.1.1 via ge-0/0/1.0
                    [BGP/170] 00:02:34, localpref 100, from 172.16.1.2
                      AS path: 76 I
                    > to 10.0.1.5 via ge-0/0/2.0
88.88.88.0/24      *[BGP/170] 00:02:29, localpref 100, from 172.16.1.2
                      AS path: 88 I
                    > to 10.0.1.5 via ge-0/0/2.0
89.89.89.0/24      *[BGP/170] 00:02:29, localpref 100, from 172.16.1.2
                      AS path: 88 I
                    > to 10.0.1.5 via ge-0/0/2.0
```

Let's take a look at R4 to see what it's receiving from R3:

```
root@r4> show bgp summary
Groups: 1 Peers: 1 Down peers: 0
Table          Tot Paths  Act Paths Suppressed    History Damp State    Pending
inet.0                 6          6          0          0          0          0
Peer                     AS      InPkt     OutPkt    OutQ   Flaps Last Up/Dwn State|#Active/
Received/Accepted/Damped...
172.16.1.3          14023         60         51       0       2       13:59 6/6/6/0
0/0/0/0

root@r4> show route protocol bgp

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

66.66.66.0/24     *[BGP/170] 00:03:12, localpref 100, from 172.16.1.3
                    AS path: 66 I
                  > to 10.0.1.9 via ge-0/0/1.0
67.67.67.0/24     *[BGP/170] 00:03:12, localpref 100, from 172.16.1.3
                    AS path: 66 I
                  > to 10.0.1.9 via ge-0/0/1.0
76.76.76.0/24     *[BGP/170] 00:03:12, localpref 100, from 172.16.1.3
                    AS path: 76 I
                  > to 10.0.1.9 via ge-0/0/1.0
77.77.77.0/24     *[BGP/170] 00:03:12, localpref 100, from 172.16.1.3
                    AS path: 76 I
                  > to 10.0.1.9 via ge-0/0/1.0
88.88.88.0/24     *[BGP/170] 00:03:44, localpref 100, from 172.16.1.3
                    AS path: 88 I
                  > to 10.0.1.13 via ge-0/0/2.0
89.89.89.0/24     *[BGP/170] 00:03:44, localpref 100, from 172.16.1.3
                    AS path: 88 I
                  > to 10.0.1.13 via ge-0/0/2.0
```

In this case, R4 is only seeing one next hop being advertised for the routes that are being reflected from R3. Specifically, the two routes for 76.76.76.0/24 and 77.77.77.0/24 that are being ingested via R1 and R2. If you want to allow for multipathing in your network and achieve better load balancing and distribution of traffic, ideally you would want to see both next hops as available options.

Let's take a further look at what R3 is advertising to better understand what is happening:

```
root@r3> show route advertising-protocol bgp 172.16.1.4

inet.0: 21 destinations, 23 routes (21 active, 0 holddown, 0 hidden)
  Prefix                Nexthop              MED     Lclpref    AS path
* 66.66.66.0/24          172.16.1.1                   100        66 I
* 67.67.67.0/24          172.16.1.1                   100        66 I
* 76.76.76.0/24          172.16.1.1                   100        76 I
* 77.77.77.0/24          172.16.1.1                   100        76 I
* 88.88.88.0/24          172.16.1.2                   100        88 I
* 89.89.89.0/24          172.16.1.2                   100        88 I
```

You can see that R3 is only advertising a single next hop for the routes of 76.76.76.0/24 and 77.77.77.0/24. In order to better understand why this is happening, let's take a deeper look inside R3's routing table:

```
root@r3> show route 76.76.76.0/24 extensive

inet.0: 21 destinations, 23 routes (21 active, 0 holddown, 0 hidden)
76.76.76.0/24 (2 entries, 1 announced)
TSI:
KRT in-kernel 76.76.76.0/24 -> {indirect(262143)} Page 0 idx 0 Type 1 val 9380f50
    Nexthop: 172.16.1.1
    Localpref: 100
    AS path: [14023] 76 I
    Communities:
    Cluster ID: 1.1.1.1
    Originator ID: 172.16.1.1
    Advertise: 00000005
Path 76.76.76.0 from 172.16.1.1 Vector len 4.  Val: 0
        *BGP    Preference: 170/-101
                Next hop type: Indirect
                Address: 0x934ca78
                Next-hop reference count: 12
                Source: 172.16.1.1
                Next hop type: Router, Next hop index: 569
                Next hop: 10.0.1.1 via ge-0/0/1.0, selected
                Protocol next hop: 172.16.1.1
                Indirect next hop: 9430570 262143
                State: <Active Int Ext>
                Local AS: 14023 Peer AS: 14023
                Age: 2:24      Metric2: 1
                Task: BGP_14023.172.16.1.1+55563
                Announcement bits (3): 0-KRT 3-BGP_RT_Background 4-Resolve tree 1
                AS path: 76 I
                Accepted
                Localpref: 100
                Router ID: 172.16.1.1
                Indirect next hops: 1
                        Protocol next hop: 172.16.1.1 Metric: 1
                Indirect next hop: 9430570 262143
                Indirect path forwarding next hops: 1
                        Next hop type: Router
                        Next hop: 10.0.1.1 via ge-0/0/1.0
                172.16.1.1/32 Originating RIB: inet.0
                  Metric: 1                     Node path count: 1
                  Forwarding nexthops: 1
                        Nexthop: 10.0.1.1 via ge-0/0/1.0
 BGP    Preference: 170/-101
        Next hop type: Indirect
        Address: 0x934c7a8
        Next-hop reference count: 8
        Source: 172.16.1.2
        Next hop type: Router, Next hop index: 571
        Next hop: 10.0.1.5 via ge-0/0/2.0, selected
        Protocol next hop: 172.16.1.2
        Indirect next hop: 9430658 262142
        State: <NotBest Int Ext>
        Inactive reason: Not Best in its group - Router ID
        Local AS: 14023 Peer AS: 14023
        Age: 3:02      Metric2: 1
        Task: BGP_14023.172.16.1.2+179
        AS path: 76 I
        Accepted
        Localpref: 100
        Router ID: 172.16.1.2
        Indirect next hops: 1
                Protocol next hop: 172.16.1.2 Metric: 1
                Indirect next hop: 9430658 262142
                Indirect path forwarding next hops: 1
                        Next hop type: Router
                        Next hop: 10.0.1.5 via ge-0/0/2.0
                172.16.1.2/32 Originating RIB: inet.0
                  Metric: 1                     Node path count: 1
                  Forwarding nexthops: 1
                        Nexthop: 10.0.1.5 via ge-0/0/2.0
```

Here, R3 is selecting the route from neighbor R1 as the preferred route and choosing to make the route from R2 inactive due to the reason of `Not Best in its group - Router ID`.

In this case, everything is behaving as expected. R3 is making a local routing decision and choosing the best routes to make active, and this is impacting the decisions on which routes to advertise to its route reflector clients. So you can see that while route reflection has some real scaling benefits, it also tends to mask some of the information in your network that would make multipathing possible.

Without the ability to have these multiple next hops available to their clients, RRs have but one available next hop at their disposal, and therefore will not be able to perform any type of load-balancing on a per-route basis.

Solution

Okay, everything we've seen so far is normal, expected BGP behavior. BGP peers advertise routes to each other in update messages and each receiver stores those routes in the routing table. For each prefix in the routing table, the routing protocol process selects a single best path, called the *active path*. In the case of route reflection, the RR is choosing to only send the single best path, or active path, to its reflector clients.

Instead of advertising only the active path to a destination, you can configure BGP to advertise multiple paths to destinations. You can take advantage of this when using route reflection so that you can once again present the availability of multiple exit points to reach a given destination. Not only does this provide for better load balancing of traffic, it provides immediate fault tolerance in the event of a particular exit point going down, as the local device will always have another option available without having to wait for the surrounding network to reconverge.

To take advantage of this ability to advertise not only the best path, but all potential paths, you need to utilize the BGP `add-path` option to your advertised NLRI. This needs to take place in two different places – both in the route reflector itself as well as with the clients.

Let's take a look at the configuration required to achieve this:

```
R3:
[edit protocols bgp]
root@r3# set group RR_Clients family inet unicast add-path send path-count 2

R4:
[edit]
root@r4# set protocols bgp group ibgp family inet unicast add-path receive
```

NOTE The `path-count` keyword indicates how many next hops you want to advertise. In this case, you are telling the route reflector to send a maximum of two paths to the RR clients, but if you want to advertise more, you can, up to a maximum of six.

Once this configuration has been committed, you can now see that R3 is advertising multiple paths to R3. And R3 is receiving multiple paths as a result:

```
R3:
root@r3> **show route advertising-protocol bgp 172.16.1.4**

inet.0: 21 destinations, 23 routes (21 active, 0 holddown, 0 hidden)
  Prefix                 Nexthop            MED     Lclpref    AS path
* 66.66.66.0/24          172.16.1.1                  100        66 I
* 67.67.67.0/24          172.16.1.1                  100        66 I
* 76.76.76.0/24         172.16.1.1               100       76 I
                     172.16.1.2               100       76 I
* 77.77.77.0/24         172.16.1.1               100       76 I
                     172.16.1.2               100       76 I
* 88.88.88.0/24         172.16.1.2               100       88 I
* 89.89.89.0/24         172.16.1.2               100       88 I

R4:
root@r4> show bgp summary
Groups: 1 Peers: 1 Down peers: 0
Table          Tot Paths  Act Paths Suppressed    History Damp State    Pending
inet.0               8          6          0          0          0          0
Peer                     AS      InPkt     OutPkt    OutQ   Flaps Last Up/Dwn State|#Active/
Received/Accepted/Damped...
172.16.1.3          14023          7          5       0       0        1:06 6/8/8/0
0/0/0/0

root@r4> show route protocol bgp

inet.0: 21 destinations, 23 routes (21 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

66.66.66.0/24      *[BGP/170] 00:01:09, localpref 100, from 172.16.1.3
                      AS path: 66 I
                    > to 10.0.1.9 via ge-0/0/1.0
67.67.67.0/24      *[BGP/170] 00:01:09, localpref 100, from 172.16.1.3
                      AS path: 66 I
                    > to 10.0.1.9 via ge-0/0/1.0
76.76.76.0/24      *[BGP/170] 00:01:09, localpref 100, from 172.16.1.3
                      AS path: 76 I
                    > to 10.0.1.9 via ge-0/0/1.0
                    [BGP/170] 00:01:09, localpref 100, from 172.16.1.3
                      AS path: 76 I
                    > to 10.0.1.13 via ge-0/0/2.0
77.77.77.0/24      *[BGP/170] 00:01:09, localpref 100, from 172.16.1.3
                      AS path: 76 I
                    > to 10.0.1.9 via ge-0/0/1.0
                    [BGP/170] 00:01:09, localpref 100, from 172.16.1.3
                      AS path: 76 I
                    > to 10.0.1.13 via ge-0/0/2.0
88.88.88.0/24      *[BGP/170] 00:01:09, localpref 100, from 172.16.1.3
                      AS path: 88 I
                    > to 10.0.1.13 via ge-0/0/2.0
89.89.89.0/24      *[BGP/170] 00:01:09, localpref 100, from 172.16.1.3
                      AS path: 88 I
                    > to 10.0.1.13 via ge-0/0/2.0
```

One thing to make note of here is that even though there's advertisement of multiple paths via our route reflector, at this point the behavior would be no different than a traditional iBGP method – even though there's multiple paths available, R4 will still only select one of them as the best path, per the BGP path selection algorithm:

```
root@r4> show route 76.76.76.0/24 exact extensive

inet.0: 21 destinations, 23 routes (21 active, 0 holddown, 0 hidden)
76.76.76.0/24 (2 entries, 1 announced)
TSI:
KRT in-kernel 76.76.76.0/24 -> {indirect(262142)}
        *BGP    Preference: 170/-101
                Next hop type: Indirect
                Address: 0x934cc28
                Next-hop reference count: 12
                Source: 172.16.1.3
                Next hop type: Router, Next hop index: 568
                Next hop: 10.0.1.9 via ge-0/0/1.0, selected
                Protocol next hop: 172.16.1.1
                Indirect next hop: 942c658 262142
                State: <Active Int Ext>
                Local AS: 14023 Peer AS: 14023
                Age: 1:47       Metric2: 1
                Task: BGP_14023.172.16.1.3+179
                Announcement bits (2): 0-KRT 4-Resolve tree 1
                AS path: 76 I (Originator) Cluster list:  1.1.1.1
                AS path:  Originator ID: 172.16.1.1
                Accepted
                Localpref: 100
                Router ID: 172.16.1.3
                Addpath Path ID: 1
                Indirect next hops: 1
                        Protocol next hop: 172.16.1.1 Metric: 1
                        Indirect next hop: 942c658 262142
                        Indirect path forwarding next hops: 1
                                Next hop type: Router
                                Next hop: 10.0.1.9 via ge-0/0/1.0
                        172.16.1.1/32 Originating RIB: inet.0
                          Metric: 1                       Node path count: 1
                          Forwarding nexthops: 1
                                Nexthop: 10.0.1.9 via ge-0/0/1.0
         BGP    Preference: 170/-101
                Next hop type: Indirect
                Address: 0x934c958
                Next-hop reference count: 8
                Source: 172.16.1.3
                Next hop type: Router, Next hop index: 569
                Next hop: 10.0.1.13 via ge-0/0/2.0, selected
                Protocol next hop: 172.16.1.2
                Indirect next hop: 942c740 262143
                State: <NotBest Int Ext>
                Inactive reason: Not Best in its group - Router ID
                Local AS: 14023 Peer AS: 14023
                Age: 1:47       Metric2: 1
                Task: BGP_14023.172.16.1.3+179
                AS path: 76 I (Originator) Cluster list:  1.1.1.1
                AS path:  Originator ID: 172.16.1.2
                Accepted
                Localpref: 100
                Router ID: 172.16.1.3
                Addpath Path ID: 2
                Indirect next hops: 1
                        Protocol next hop: 172.16.1.2 Metric: 1
                        Indirect next hop: 942c740 262143
                        Indirect path forwarding next hops: 1
                                Next hop type: Router
                                Next hop: 10.0.1.13 via ge-0/0/2.0
                        172.16.1.2/32 Originating RIB: inet.0
                          Metric: 1                       Node path count: 1
                          Forwarding nexthops: 1
                                Nexthop: 10.0.1.13 via ge-0/0/2.0
```

Here you can see that R4 is selecting the path via R1 as the best path based on the router ID selection. Next, you can go on to enable true multi-pathing by enabling the `multipath` statement within BGP and also enabling a forwarding table load-balancing policy:

```
R4:
[edit]
root@r4# set protocols bgp group ibgp multipath

[edit]
root@r4# set policy-options policy-statement load-balance then load-balance per-packet

[edit routing-options]
root@r4# set routing-options forwarding-table export load-balance
```

Once this configuration has been committed, you can see that multiple next hops are available not only in the routing table, but also in the forwarding table, too:

```
root@r4> show bgp summary
Groups: 1 Peers: 1 Down peers: 0
Table          Tot Paths  Act Paths Suppressed    History Damp State    Pending
inet.0                8          8          0          0          0          0
Peer                     AS      InPkt     OutPkt    OutQ   Flaps Last Up/Dwn State|#Active/
Received/Accepted/Damped...
172.16.1.3            14023         38         37       0       0       15:30 8/8/8/0
0/0/0/0

root@r4> show route protocol bgp

inet.0: 21 destinations, 23 routes (21 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

66.66.66.0/24      *[BGP/170] 00:15:38, localpref 100, from 172.16.1.3
                      AS path: 66 I
                    > to 10.0.1.9 via ge-0/0/1.0
67.67.67.0/24      *[BGP/170] 00:15:38, localpref 100, from 172.16.1.3
                      AS path: 66 I
                    > to 10.0.1.9 via ge-0/0/1.0
76.76.76.0/24      *[BGP/170] 00:15:38, localpref 100, from 172.16.1.3
                      AS path: 76 I
```

```
                  to 10.0.1.9 via ge-0/0/1.0
                > to 10.0.1.13 via ge-0/0/2.0
                [BGP/170] 00:15:38, localpref 100, from 172.16.1.3
                  AS path: 76 I
                > to 10.0.1.13 via ge-0/0/2.0
77.77.77.0/24     *[BGP/170] 00:15:38, localpref 100, from 172.16.1.3
                  AS path: 76 I
                  to 10.0.1.9 via ge-0/0/1.0
                > to 10.0.1.13 via ge-0/0/2.0
                [BGP/170] 00:15:38, localpref 100, from 172.16.1.3
                  AS path: 76 I
                > to 10.0.1.13 via ge-0/0/2.0
88.88.88.0/24     *[BGP/170] 00:15:38, localpref 100, from 172.16.1.3
                  AS path: 88 I
                > to 10.0.1.13 via ge-0/0/2.0
89.89.89.0/24     *[BGP/170] 00:15:38, localpref 100, from 172.16.1.3
                  AS path: 88 I
                > to 10.0.1.13 via ge-0/0/2.0

root@r4> show route 76.76.76.0/24 exact extensive

inet.0: 21 destinations, 23 routes (21 active, 0 holddown, 0 hidden)
76.76.76.0/24 (2 entries, 1 announced)
TSI:
KRT in-kernel 76.76.76.0/24 -> {indirect(262142), indirect(262143)}
        *BGP    Preference: 170/-101
                Next hop type: Indirect
                Address: 0x94281b0
                Next-hop reference count: 6
                Source: 172.16.1.3
                Next hop type: Router, Next hop index: 568
                Next hop: 10.0.1.9 via ge-0/0/1.0
                Next hop type: Router, Next hop index: 569
                Next hop: 10.0.1.13 via ge-0/0/2.0, selected
                Protocol next hop: 172.16.1.1
                Indirect next hop: 942c658 262142
                Protocol next hop: 172.16.1.2
                Indirect next hop: 942c740 262143
                State: <Active Int Ext>
                Local AS: 14023 Peer AS: 14023
                Age: 15:49     Metric2: 1
                Task: BGP_14023.172.16.1.3+179
                Announcement bits (2): 0-KRT 4-Resolve tree 1
                AS path: 76 I (Originator) Cluster list:  1.1.1.1
                AS path:  Originator ID: 172.16.1.1
                Accepted Multipath
                Localpref: 100
                Router ID: 172.16.1.3
                Addpath Path ID: 1
                Indirect next hops: 2
                        Protocol next hop: 172.16.1.1 Metric: 1
                        Indirect next hop: 942c658 262142
                        Indirect path forwarding next hops: 1
                                Next hop type: Router
                                Next hop: 10.0.1.9 via ge-0/0/1.0
                        172.16.1.1/32 Originating RIB: inet.0
                          Metric: 1                    Node path count: 1
                          Forwarding nexthops: 1
                                Nexthop: 10.0.1.9 via ge-0/0/1.0
                        Protocol next hop: 172.16.1.2 Metric: 1
                        Indirect next hop: 942c740 262143
                        Indirect path forwarding next hops: 1
                                Next hop type: Router
                                Next hop: 10.0.1.13 via ge-0/0/2.0
                        172.16.1.2/32 Originating RIB: inet.0
                          Metric: 1                    Node path count: 1
                          Forwarding nexthops: 1
                                Nexthop: 10.0.1.13 via ge-0/0/2.0
             BGP    Preference: 170/-101
                Next hop type: Indirect
                Address: 0x934c958
                Next-hop reference count: 8
                Source: 172.16.1.3
                Next hop type: Router, Next hop index: 569
                Next hop: 10.0.1.13 via ge-0/0/2.0, selected
                Protocol next hop: 172.16.1.2
                Indirect next hop: 942c740 262143
                State: <NotBest Int Ext>
                Inactive reason: Not Best in its group - Router ID
                Local AS: 14023 Peer AS: 14023
                Age: 15:49      Metric2: 1
                Task: BGP_14023.172.16.1.3+179
                AS path: 76 I (Originator) Cluster list:  1.1.1.1
                AS path:  Originator ID: 172.16.1.2
                Accepted MultipathContrib
                Localpref: 100
                Router ID: 172.16.1.3
                Addpath Path ID: 2
                Indirect next hops: 1
                        Protocol next hop: 172.16.1.2 Metric: 1
                        Indirect next hop: 942c740 262143
                        Indirect path forwarding next hops: 1
                                Next hop type: Router
                                Next hop: 10.0.1.13 via ge-0/0/2.0
                        172.16.1.2/32 Originating RIB: inet.0
                          Metric: 1                    Node path count: 1
                          Forwarding nexthops: 1
                                Nexthop: 10.0.1.13 via ge-0/0/2.0


root@r4> show route forwarding-table destination 76.76.76.0/24
Routing table: default.inet
Internet:
Destination        Type RtRef Next hop           Type Index NhRef Netif
76.76.76.0/24      user     0                    ulst 262146     2
                                             indr 262142     4
                              10.0.1.9          ucst   568     5 ge-0/0/1.0
```

```
                               indr 262143    4
                  10.0.1.13        ucst   569    5 ge-0/0/2.0

Routing table: __master.anon__.inet
Internet:
Destination       Type RtRef Next hop         Type Index NhRef Netif
default          perm    0                 rjct   524    1
```

Discussion

The add-path attribute is a particularly useful tool to use, especially when implementing route reflection in a given environment. It enables an RR to advertise not only the best path, but multiple paths to a given destination. There are many other use cases outside of simple route reflection where the add-path attribute may be desired, however, especially where multiple paths are needed to be instantiated in the routing table. It is important to note that in order to enable true multipathing, you must still enable the `multi-path` knob within BGP and also instantiate a forwarding table load-balancing policy.

Another consideration is that this knob becomes very important when designing IP Clos Fabrics employing the use of route reflection. In these cases, you may have a number of destinations available in your IP fabric in order to reach VTEP endpoints for creating VXLAN overlays. Using route reflection simplifies the design of the underlay network and allows the underlay to scale, but without the add-path attribute, you'll lose the ability to multipath through the network.

Recipe 4: EVPN and Virtual Machine Mobility

As hardware and services become more virtualized your network needs to support these services. For example, connecting Layer 2 networks across multiple datacenters continues to be a pressing challenge. In addition to avoiding the common pitfalls of stretching Layer 2 across the WAN, forcing updates for MAC address learning in a single Layer 2 domain becomes problematic. One solution to these issues is to utilize Virtual Extensible LAN (VXLAN) technologies with Ethernet VPN (EVPN) signaling. Let's investigate this as a cookbook recipe.

Problem

Let's say you have two data centers, one in Boston and one in Chicago, that have virtualized infrastructures as shown in Figure 4.1. You want to extend a VLAN across a dark fiber circuit that connects the two data centers together. In addition, the data centers will utilize vMotion to transparently migrate Virtual Machine (VM) guests from one data center to the other. And you want to avoid a pure Layer 2 stretch due to inherent problems with stability in the event of a loop on the network. How can these requirements be accomplished?

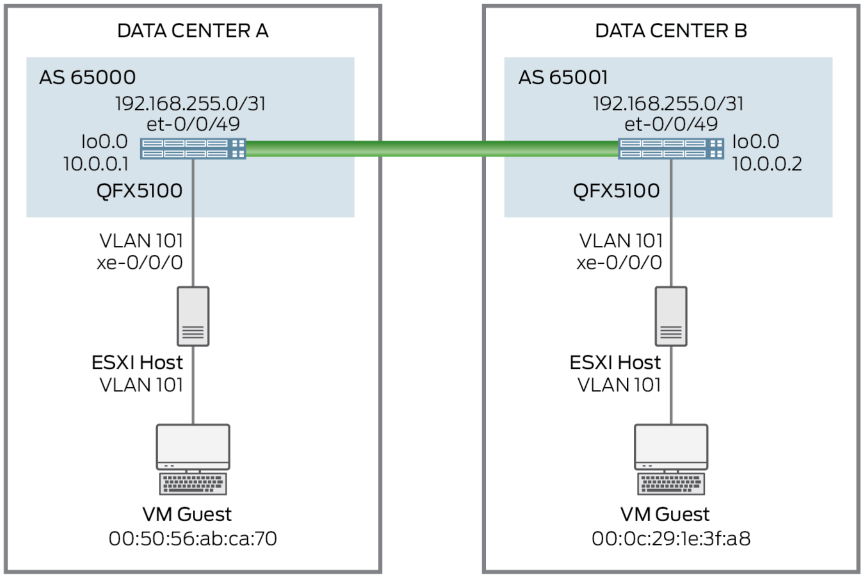


Figure 4.1 This Recipe's Topology of Two Data Centers

Solution

To solve this quandary let's use EVPN with a VXLAN overlay using QFX5100 switches. EVPN becomes the control plane and source of all MAC addresses, and can use *proxy arp* for hosts external to the local virtualized environment. EVPN also has built-in protections that reduce broadcast storms in the event of a loop inside a data center. When VM guests are migrated from one data center to another, EVPN can send updates via BGP to force MAC learning across each respective data center. VXLAN provides the overlay infrastructure to transport traffic across the data center.

NOTE The QFX5100 switches are only capable of Layer 2 Virtual Tunneling Endpoints (VTEPs).

Configuration

First, let's configure the interfaces on the QFX5100 switches. The interface xe-0/0/0 will be bridged to the vCenter hosts, while et-0/0/49 will be used to terminate the fiber connecting the two data centers. The loopback interface (lo0.0) will be utilized as a VTEP as well as the route distinguisher:

```
## QFX5100-Boston
[edit]
set interfaces xe-0/0/0 description “To vCenter”
set interfaces xe-0/0/0 encapsulation ethernet-bridge
set interfaces xe-0/0/0 unit 0
set interfaces et-0/0/49 description “To Chicago QFX”
set interfaces et-0/0/49 mtu 1600
set interfaces et-0/0/49 unit 0 family inet address 192.168.255.1/31
set interfaces lo0 unit 0 family inet address 10.0.0.2/32

## QFX5100-Chicago
[edit]
set interfaces xe-0/0/0 description “To vCenter”
set interfaces xe-0/0/0 encapsulation ethernet-bridge
set interfaces xe-0/0/0 unit 0
set interfaces et-0/0/49 description “To Boston QFX”
set interfaces et-0/0/49 mtu 1600
set interfaces et-0/0/49 unit 0 family inet address 192.168.255.0/31
set interfaces lo0 unit 0 family inet address 10.0.0.1/32
```

NOTE The interfaces connecting the two data centers must have a Maximum Transmission Unit (MTU) be set to at least 1550 due to the additional VXLAN header for VXLAN-encapsulated traffic. A general recommendation is to set this to 1600 for future growth in VXLAN or to accommodate future headers.

Next you need to to configure a VLAN that will be tunneled across the network:

```
## QFX5100-Boston and QFX5100-Chicago
[edit]
set vlans vlan101 interface xe-0/0/0.0
```

Before BGP is configured you need to set up several policies to allow each QFX to reach the other’s loopback address. For the sake of simplicity, the policy will advertise all directly connected routes. In addition to this you need to configure a community that will be tagged on each advertised EVPN route, and an import policy to accept all routes learned through that community:

```
## QFX5100-Boston and QFX5100-Chicago
[edit]
set policy-options policy-statement VRF-IMPORT term vxlan101 from community vxlan101
set policy-options policy-statement VRF-IMPORT term vxlan101 then accept
set policy-options policy-statement VRF-IMPORT then reject
set policy-options policy-statement allow-loopbacks term direct from protocol direct
set policy-options policy-statement allow-loopbacks term direct then accept
set policy-options policy-statement allow-loopbacks then reject
set policy-options community vxlan101 members target:1:101
```

Next up is the BGP configuration. Since BGP supports the use of multiple protocols (in this case, IPv4 and EVPN) you can configure two families to be transferred in a single BGP session. Both sites will use a unique Autonomous System (AS) to ensure that External BGP (eBGP) is used. This is also where the export policy `allow-loopbacks` comes into play:

```
## QFX5100-Boston
[edit]
set routing-options router-id 10.0.0.2
set routing-options autonomous-system 65002
set protocols bgp group vxlan type external
set protocols bgp group vxlan family inet unicast
set protocols bgp group vxlan family evpn signaling
set protocols bgp group vxlan export allow-loopbacks
set protocols bgp group vxlan neighbor 192.168.255.0 description qfx5100-chicago
set protocols bgp group vxlan neighbor 192.168.255.0 peer-as 65001

## QFX5100-Chicago
[edit]
set routing-options router-id 10.0.0.1
set routing-options autonomous-system 65001
set protocols bgp group vxlan type external
set protocols bgp group vxlan family inet unicast
set protocols bgp group vxlan family evpn signaling
set protocols bgp group vxlan export allow-loopbacks
set protocols bgp group vxlan neighbor 192.168.255.1 description qfx5100-boston
set protocols bgp group vxlan neighbor 192.168.255.1 peer-as 65002
```

Okay, now the VXLAN configuration. Each VLAN is tied to a unique Virtual Network Identifier (VNI) so you can transport multiple VLANs across a single BGP session. Each VNI is then tied to a unique Virtual Routing and Forwarding (VRF) instance in order to set up a path for broadcast, unknown unicast, and multicast (BUM) traffic between the QFX Series 5100 devices. Each VRF will import the routes from the import policy created in a previous step. Since you want to dynamically add more QFX switches for data center expansion in the future, this VRF signaling will allow new QFX Series devices to automatically join the topology. The VRFs also provide isolation so routes from other VNIs are never leaked into VNI101. Each QFX will have a `route-distinguisher` configured so you can distinguish where learned routes originated. From here, let's configure the VTEP source interface as lo0.0. This serves as the point where LAN traffic is encapsulated in a VXLAN header before it is transported between data centers:

```
## QFX5100-Boston
[edit]
set switch-options vtep-source-interface lo0.0
set switch-options route-distinguisher 10.0.0.2:1
set switch-options vrf-import VRF-IMPORT
set switch-options vrf-target target:9999:9999
set vlans vlan101 vxlan vni 101
set vlans vlan101 vxlan ingress-node-replication

## QFX5100-Chicago
[edit]
set switch-options vtep-source-interface lo0.0
set switch-options route-distinguisher 10.0.0.1:1
set switch-options vrf-import VRF-IMPORT
set switch-options vrf-target target:9999:9999
set vlans vlan101 vxlan vni 101
set vlans vlan101 vxlan ingress-node-replication
```

Finally, let's configure the EVPN signaling and VXLAN encapsulation. Each EVPN route learned inside VNI101 will be exported with a VRF target called target:1:101, which matches the community that was configured in the previous steps. All routes matching this target will then be accepted by the VRF due to the VRF-IMPORT policy that was previously configured. Then a VNI list that will be allowed through this EVPN signaling is set up:

```
## QFX5100-Boston and QFX5100-Chicago
[edit]
set protocols evpn vni-options vni 101 vrf-target export target:1:101
set protocols evpn encapsulation vxlan
set protocols evpn extended-vni-list all
set protocols evpn multicast-mode ingress-replication
```

Commit the changes and then you can start to verify the configuration:

```
## QFX5100-Boston and QFX5100-Chicago
[edit]
root@# commit and-quit
configuration check succeeds
commit complete
```

Verification

You should now have a fully functional VXLAN deployment that allows you to migrate VM guests from one data center to another. Let's begin with some BGP validation:

```
root@Boston-QFX5100> show bgp summary
Groups: 1 Peers: 1 Down peers: 0
Table          Tot Paths  Act Paths Suppressed    History Damp State    Pending
inet.0
                       2          1          0          0          0          0
bgp.evpn.0
                       2          2          0          0          0          0
Peer                     AS      InPkt     OutPkt    OutQ   Flaps Last Up/Dwn State|#Active/
Received/Accepted/Damped...
192.168.255.0        65001       2642       2794       0       0    19:48:39 Establ
  inet.0: 1/2/2/0
  bgp.evpn.0: 2/2/2/0
  default-switch.evpn.0: 2/2/2/0
  __default_evpn__.evpn.0: 0/0/0/0

root@Chicago-QFX5100> show bgp summary
Groups: 1 Peers: 1 Down peers: 0
Table          Tot Paths  Act Paths Suppressed    History Damp State    Pending
inet.0
                       2          1          0          0          0          0
bgp.evpn.0
                       1          1          0          0          0          0
Peer                     AS      InPkt     OutPkt    OutQ   Flaps Last Up/Dwn State|#Active/
Received/Accepted/Damped...
192.168.255.1       65002      2796      2642      0      3   19:48:46 Establ
 inet.0: 1/2/2/0
 bgp.evpn.0: 1/1/1/0
```

```
  default-switch.evpn.0: 1/1/1/0
  __default_evpn__.evpn.0: 0/0/0/0
```

You can see in the output that both the family inet and family evpn protocols are being accepted across the BGP peers. Now let's look at the Ethernet switching table to determine the VM guests that are being learned from vCenter:

```
root@Boston-QFX5100> show ethernet-switching table interface xe-0/0/0.0

MAC database for interface xe-0/0/0.0

MAC flags (S - static MAC, D - dynamic MAC, L - locally learned, P - Persistent static
           SE - statistics enabled, NM - non configured MAC, R - remote PE MAC, O - ovsdb MAC)

Ethernet switching table : 3 entries, 3 learned
Routing instance : default-switch
   Vlan              MAC               MAC     Logical              Active
   name              address            flags   interface             source
   vlan101            00:0c:29:1e:3f:a8   D       xe-0/0/0.0

root@Chicago-QFX5100> show ethernet-switching table interface xe-0/0/0.0

MAC database for interface xe-0/0/0.0

MAC flags (S - static MAC, D - dynamic MAC, L - locally learned, P - Persistent static
           SE - statistics enabled, NM - non configured MAC, R - remote PE MAC, O - ovsdb MAC)


Ethernet switching table : 3 entries, 3 learned
Routing instance : default-switch
   Vlan              MAC               MAC     Logical              Active
   name              address            flags   interface             source
   vlan101            00:50:56:ab:ca:70   D       xe-0/0/0.0
```

From this output you now know that Boston-QFX5100 has learned MAC Address 00:0c:29:1e:3f:a8 on its xe-0/0/0 interface, while Chicago-QFX5100 has learned MAC Address 00:50:56:ab:ca:70 from its xe-0/0/0 interface. What do these learned MACs look like on the remote switch?

```
root@Boston-QFX5100> show ethernet-switching table 00:50:56:ab:ca:70

MAC flags (S - static MAC, D - dynamic MAC, L - locally learned, P - Persistent static
           SE - statistics enabled, NM - non configured MAC, R - remote PE MAC, O - ovsdb MAC)


Ethernet switching table : 3 entries, 3 learned
Routing instance : default-switch
   Vlan              MAC               MAC     Logical              Active
   name              address            flags   interface             source
   vlan101            00:50:56:ab:ca:70   D       vtep.32769             10.0.0.1

root@Chicago-QFX5100> show ethernet-switching table 00:0c:29:1e:3f:a8

MAC flags (S - static MAC, D - dynamic MAC, L - locally learned, P - Persistent static
           SE - statistics enabled, NM - non configured MAC, R - remote PE MAC, O - ovsdb MAC)
```

```
Ethernet switching table : 3 entries, 3 learned
Routing instance : default-switch
   Vlan             MAC              MAC     Logical            Active
   name             address           flags   interface           source
   vlan101           00:0c:29:1e:3f:a8   D      vtep.32769          10.0.0.2
```

From this output you can verify the MACs are being learned remotely through the VTEP interfaces, which are sourced by the IP address of the loopback interface of the respective remote switch. Now let's look at the routes being learned by the EVPN protocol:

```
root@Boston-QFX5100> show route protocol bgp table bgp.evpn.0

bgp.evpn.0: 4 destinations, 4 routes (4 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

2:10.0.0.1:1::101::00:50:56:ab:ca:70/304
                   *[BGP/170] 20:37:40, localpref 100
                      AS path: 65001 I, validation-state: unverified
                    > to 192.168.255.0 via et-0/0/49.0
3:10.0.0.1:1::101::10.0.0.1/304
                   *[BGP/170] 20:37:40, localpref 100
                      AS path: 65001 I, validation-state: unverified
                    > to 192.168.255.0 via et-0/0/49.0

root@Chicago-QFX5100> show route protocol bgp table bgp.evpn.0

bgp.evpn.0: 4 destinations, 4 routes (4 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

2:10.0.0.2:1::101::00:0c:29:1e:3f:a8/304
                   *[BGP/170] 00:19:52, localpref 100
                      AS path: 65002 I, validation-state: unverified
                    > to 192.168.255.1 via et-0/0/49.0
3:10.0.0.2:1::101::10.0.0.2/304
                   *[BGP/170] 20:37:58, localpref 100
                      AS path: 65002 I, validation-state: unverified
                    > to 192.168.255.1 via et-0/0/49.0
```

Each route shows the following information listed in Table 4.1.

Table 4.1 EVPN Route Breakdown

Switch	Route Type	VTEP	VNI	MAC Address	Gateway IP	Preference
Boston-QFX5100	2 (MAC Address)	10.0.0.1	101	00:50:56:ab:ca:70	--	304
Boston-QFX5100	3 (BUM Path)	10.0.0.1	101	--	10.0.0.1	304
Chicago-QFX5100	2 (MAC Address)	10.0.0.2	101	00:0c:29:1e:3f:a8	--	304
Chicago-QFX5100	3 (BUM Path)	10.0.0.2	101	--	10.0.0.2	304

What happens when a VM is migrated from one data center to another? Let's find out! Let's vMotion the VM Guest 00:0c:29:1e:3f:a8 from the Boston Data Center to the Chicago Data Center and review the Ethernet switching table and the EVPN routes on the Boston QFX5100 Series:

```
root@Boston-QFX5100> show ethernet-switching table 00:0c:29:1e:3f:a8

MAC flags (S - static MAC, D - dynamic MAC, L - locally learned, P - Persistent static
         SE - statistics enabled, NM - non configured MAC, R - remote PE MAC, O - ovsdb MAC)

Ethernet switching table : 4 entries, 4 learned
Routing instance : default-switch
   Vlan              MAC               MAC     Logical             Active
   name              address            flags   interface            source
   vlan101           00:0c:29:1e:3f:a8   D       vtep.32769           10.0.0.1

root@Boston-QFX5100> show route protocol bgp table bgp.evpn.0

bgp.evpn.0: 6 destinations, 6 routes (6 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

2:10.0.0.1:1::101::00:0c:29:1e:3f:a8/304
                   *[BGP/170] 00:01:01, localpref 100
                      AS path: 65001 I, validation-state: unverified
                    > to 192.168.255.0 via et-0/0/49.0
2:10.0.0.1:1::101::00:50:56:ab:ca:70/304
                   *[BGP/170] 20:57:17, localpref 100
                      AS path: 65001 I, validation-state: unverified
                    > to 192.168.255.0 via et-0/0/49.0
3:10.0.0.1:1::101::10.0.0.1/304
                   *[BGP/170] 20:57:17, localpref 100
                      AS path: 65001 I, validation-state: unverified
                    > to 192.168.255.0 via et-0/0/49.0
```

You can see the MAC address that was originally learned on xe-0/0/0 is now being advertised as an EVPN route, and the Ethernet switching table now shows the VTEP interface as the source for the MAC Address! Any VM Guests on the Boston QFX5100 that need to reach 00:0c:29:1e:3f:a8 will be proxied through the QFX5100 Series.

Discussion

Using EVPN and VXLAN together is a very effective tool for handling data center interconnects without having to stretch Layer 2 protocols between the data centers. EVPN provides the signaling capabilities for learning MAC addresses over a Layer-3 protocol, while the VXLAN overlay handles the transport of communication between VM Guests.

MORE? Try the excellent *Day One: Using Ethernet VPNs for Data Center Interconnect,* a more thorough topic discussion of using EVPN: http://www.juniper.net/us/en/training/jnbooks/day-one/proof-concept-labs/using-ethernet-vpns/. For an excellent overview of EVPN go to the Juniper TechLibrary: https://www.juniper.net/documentation/en_US/junos/topics/concept/evpns-overview.html.

Recipe 5:
OSPF as a PE-CE Routing Protocol in MPLS VPNs

While the OSPF areas in a network can correspond to individual sites, there are other situations when multiple sites of a network are required to be in a single area or domain. In these situations, you can opt for a VPN service from a service provider and exchange the routing information between sites using an MPLS VPN backbone.

Problem

With traditional OSPF to BGP redistribution, the OSPF route type is not preserved. So even if the same route is redistributed again in OSPF, via BGP, it always appears to be an external route (LSA Type 5). This problem can be dealt with in two ways:

- Using a sham link
- Using a BGP extended community domain ID and having MPLS L3VPN extend the typical OSPF routes to multiple CE sites

Both solutions are covered in this Ambassador Cookbook recipe.

Solution 1: Sham link

A sham link can be used to connect the VPN client sites running the OSPF protocol. A sham link is an intra-area logical point-to-point link between two PE VRFs and can be used when a backdoor link exists between the two sites (it's supposed to be a backup link while traffic through the MPLS VPN is primary).

The backup and primary selection between the sham link and back-door link can be manipulated through metric change. Through a sham link, the OSPF packets are tunneled across MPLS LSPs between the PE routers. A receiving PE router then sends the received MPLS-tunneled OSPF LSAs to the local CE router, hence the OSPF LSA type is retained. The routes learned by the remote PE router through the OSPF protocol (the sham link) cannot be used by the PE router for forwarding, hence PE routers also need to learn those routes through MP-BGP.

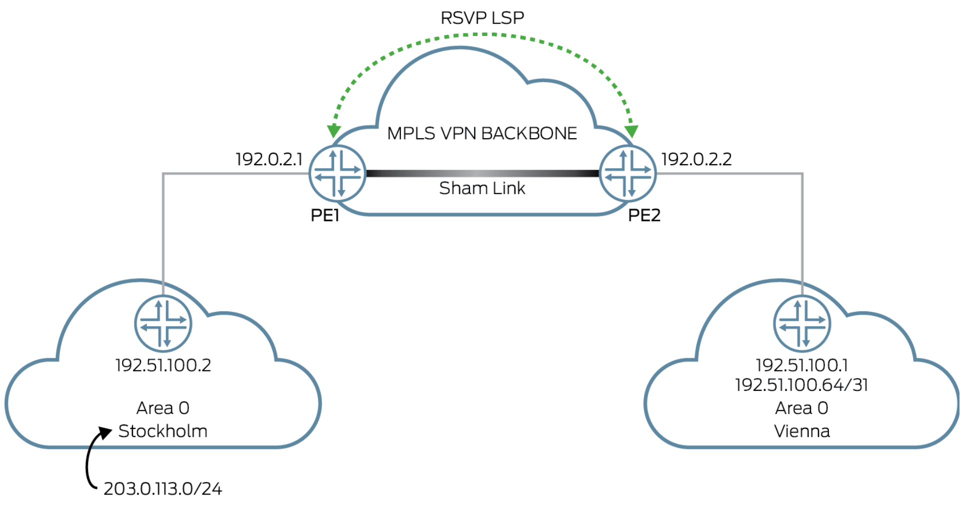


Figure 5.1 This Recipe's Sham Link Solution Topology

Figure 5.1 shows the sham link functionality. The first task is to get basic topology set, getting MP-BGP established between the PE routers (PE1 and PE2) and the MPLS LSPs in place, then configuring the routing instance VRFs on both PE1 and PE2.

NOTE The OSPF protocol is already configured on both sites, while subnet 203.0.113.0/24 is an external subnet to Stockholm CE.

To configure OSPF sham link:

Configure MX Series PE1 and PE2:

```
lab@STOCKHOLM> show configuration protocols ospf
area 0.0.0.0 {
    interface ge-0/2/2.0;                           >> Interface connected to PE1
    interface lo0.0;
}

lab@VIENNA> show configuration protocols ospf
area 0.0.0.0 {
    interface ge-0/2/4.0;                           >> Interface connected to PE2
    interface lo0.0;
    interface ge-0/2/5.0;     >> Interface connected to another OSPF speaker in VIENNA cloud
}
```

Configure the MX Series PE routers:

```
lab@PE1> show configuration routing-instances STOCKHOLM_VRF
instance-type vrf;
interface ge-0/2/1.0;
interface lo0.1;
route-distinguisher 192.0.2.1:64513;
vrf-target target:123:100;
protocols {
    ospf {
        sham-link local 192.0.2.1;
        area 0.0.0.0 {
            sham-link-remote 192.0.2.2 metric 1;
            interface ge-0/2/1.0;
            interface lo0.1;
        }
    }
}

lab@PE2> show configuration routing-instances VIENNA_VRF
instance-type vrf;
interface ge-0/2/1.0;
interface lo0.1;
route-distinguisher 192.0.2.2:64512;
vrf-target target:123:100;
protocols {
    ospf {
        sham-link local 192.0.2.2;
        area 0.0.0.0 {
            sham-link-remote 192.0.2.1 metric 1;
            interface lo0.1;
            interface ge-0/2/1.0;
        }
    }
}
```

As mentioned, the remote PE routers won't be able to forward packets through LSAs learned from the sham link, so you need to have "vrf-target" or "vrf-export/import policies" in place. These will handle route distribution through MP-BGP. The sham link's local address should be the loopback address and the remote PE's sham link will be the loopback address of the concerned PE. The MP-BGP reachability of loopback addresses between PE routers is taken care by the vrf-target or vrf-export/import policies.

To verify the sham link:

First verify the OSPF adjacency:

```
lab@STOCKHOLM> show ospf neighbor
Address          Interface              State     ID               Pri  Dead
192.51.100.60    ge-0/2/2.0             Full      192.0.2.1        128    36

lab@VIENNA> show ospf neighbor
Address          Interface              State     ID                Pri  Dead
192.51.100.50    ge-0/2/4.0             Full      192.0.2.2         128    34
192.51.100.65    ge-0/2/5.0             Full      192.51.100.65    128    33
```

```
lab@PE2> show ospf neighbor instance VIENNA_VRF
Address          Interface              State     ID               Pri  Dead
192.51.100.51    ge-0/2/1.0             Full      192.51.100.1     128    37
192.0.2.1        shamlink.0             Full      192.0.2.1          0    35

lab@PE1> show ospf neighbor instance STOCKHOLM_VRF
Address          Interface              State     ID               Pri  Dead
192.51.100.61    ge-0/2/1.0             Full      192.51.100.2     128    33
192.0.2.2        shamlink.0             Full      192.0.2.2          0    39
```

Now check the OSPF database status on both MX routers:

```
lab@STOCKHOLM> show ospf database

    OSPF database, Area 0.0.0.0
 Type       ID               Adv Rtr           Seq      Age  Opt  Cksum  Len
Router   192.0.2.1        192.0.2.1        0x8000000a   305  0x22 0x76c2  48
Router   192.0.2.2        192.0.2.2        0x80000006   335  0x22 0x86b2  48
Router   192.51.100.1     192.51.100.1     0x80000006   616  0x22 0xab0d  60
Router  *192.51.100.2     192.51.100.2     0x80000008   304  0x22 0xcb2b  48
Router   192.51.100.65    192.51.100.65    0x80000003   612  0x22 0x804a  36
Network *192.51.100.61    192.51.100.2     0x80000002  2430  0x22 0xe0d6  32
Network  192.51.100.51    192.51.100.1     0x80000001  1700  0x22 0xe255  32
Network  192.51.100.65    192.51.100.65    0x80000001   617  0x22 0xae66  32
    OSPF AS SCOPE link state database
 Type       ID               Adv Rtr           Seq      Age  Opt  Cksum  Len
Extern  *203.0.113.0      192.51.100.2     0x80000001   344  0x22 0x9c7e  36

lab@VIENNA> show ospf database

    OSPF database, Area 0.0.0.0
 Type       ID               Adv Rtr           Seq      Age  Opt  Cksum  Len
Router   192.0.2.1        192.0.2.1        0x8000000a   688  0x22 0x76c2  48
Router   192.0.2.2        192.0.2.2        0x80000006   716  0x22 0x86b2  48
Router  *192.51.100.1     192.51.100.1     0x80000006   995  0x22 0xab0d  60
Router   192.51.100.2     192.51.100.2     0x80000008   689  0x22 0xcb2b  48
Router   192.51.100.65    192.51.100.65    0x80000003   991  0x22 0x804a  36
Network  192.51.100.61    192.51.100.2     0x80000002  2815  0x22 0xe0d6  32
Network *192.51.100.51    192.51.100.1     0x80000001  2079  0x22 0xe255  32
Network  192.51.100.65    192.51.100.65    0x80000001   996  0x22 0xae66  32
    OSPF AS SCOPE link state database
 Type       ID               Adv Rtr           Seq      Age  Opt  Cksum  Len
Extern   203.0.113.0      192.51.100.2     0x80000001   729  0x22 0x9c7e  36
```

You can see that Stockholm retains the LSA type (Router/Network) learned from Vienna as highlighted in boldface. Even the External LSA being advertised from Stockholm to Vienna is installed in the database of Vienna as an External LSA Type 5.

Now let's check out the route table on the MXs:

```
lab@STOCKHOLM> show route table inet.0

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

192.51.100.60/31    *[Direct/0] 00:41:39
                     > via ge-0/2/2.0
192.51.100.61/32    *[Local/0] 00:41:39
                       Local via ge-0/2/2.0
192.51.100.1/32     *[OSPF/10] 00:06:02, metric 3
```

```
                    > to 192.51.100.60 via ge-0/2/2.0
192.51.100.2/32    *[Direct/0] 00:41:40
                    > via lo0.0
192.51.100.50/31   *[OSPF/10] 00:06:02, metric 3
                    > to 192.51.100.60 via ge-0/2/2.0
192.51.100.64/31   *[OSPF/10] 00:06:02, metric 4
                    > to 192.51.100.60 via ge-0/2/2.0
203.0.113.0/24     *[Static/5] 00:06:13
                      Receive
224.0.0.5/32       *[OSPF/10] 01:13:38, metric 1
                      MultiRecv

lab@VIENNA> show route table inet.0

inet.0: 9 destinations, 9 routes (9 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

192.51.100.60/31   *[OSPF/10] 00:11:33, metric 3
                    > to 192.51.100.50 via ge-0/2/4.0
192.51.100.1/32    *[Direct/0] 00:34:58
                    > via lo0.0
192.51.100.2/32    *[OSPF/10] 00:11:33, metric 3
                    > to 192.51.100.50 via ge-0/2/4.0
192.51.100.50/31   *[Direct/0] 00:34:58
                    > via ge-0/2/4.0
192.51.100.51/32   *[Local/0] 00:34:58
                      Local via ge-0/2/4.0
192.51.100.64/31   *[Direct/0] 00:16:54
                    > via ge-0/2/5.0
192.51.100.64/32   *[Local/0] 00:16:54
                      Local via ge-0/2/5.0
203.0.113.0/24     *[OSPF/150] 00:11:33, metric 0, tag 0
                    > to 192.51.100.50 via ge-0/2/4.0
224.0.0.5/32       *[OSPF/10] 01:18:43, metric 1
                      MultiRecv
```

And now let’s check the routes in the PE VRFs:

```
lab@PE1> show route table STOCKHOLM_VRF.inet.0

STOCKHOLM_VRF.inet.0: 10 destinations, 13 routes (10 active, 0 holddown, 3 hidden)
+ = Active Route, - = Last Active, * = Both

192.51.100.60/31   *[Direct/0] 00:41:39
                    > via ge-0/2/1.0
192.51.100.60/32   *[Local/0] 00:41:39
                      Local via ge-0/2/1.0
192.0.2.1/32       *[Direct/0] 00:41:39
                    > via lo0.1
192.0.2.2/32       *[BGP/170] 00:25:10, localpref 100, from 192.0.2.2
                      AS path: 64512 I
                    > to 192.0.2.129 via ge-0/2/0.0, label-switched-path PE1_PE2
192.51.100.1/32    *[BGP/170] 00:25:10, MED 1, localpref 100, from 192.0.2.2
                      AS path: 64512 I
                    > to 192.0.2.129 via ge-0/2/0.0, label-switched-path PE1_PE2
192.51.100.2/32    *[OSPF/10] 00:06:08, metric 1
                    > to 192.51.100.61 via ge-0/2/1.0
192.51.100.50/31   *[BGP/170] 00:25:10, localpref 100, from 192.0.2.2
                      AS path: 64512 I
                    > to 192.0.2.129 via ge-0/2/0.0, label-switched-path PE1_PE2
192.51.100.64/31   *[BGP/170] 00:10:40, MED 2, localpref 100, from 192.0.2.2
                      AS path: 64512 I
                    > to 192.0.2.129 via ge-0/2/0.0, label-switched-path PE1_PE2
203.0.113.0/24     *[OSPF/150] 00:06:13, metric 0, tag 0
```

```
                       > to 192.51.100.61 via ge-0/2/1.0
224.0.0.5/32          *[OSPF/10] 00:41:40, metric 1
                         MultiRecv

lab@PE2> show route table VIENNA_VRF.inet.0

VIENNA_VRF.inet.0: 10 destinations, 13 routes (10 active, 0 holddown, 3 hidden)
+ = Active Route, - = Last Active, * = Both

192.51.100.60/31      *[BGP/170] 00:11:44, localpref 100, from 192.0.2.1
                         AS path: 64513 I, validation-state: unverified
                       > to 192.0.2.160 via ge-0/2/3.0, label-switched-path PE2_PE1
192.0.2.1/32          *[BGP/170] 00:30:53, localpref 100, from 192.0.2.1
                         AS path: 64513 I, validation-state: unverified
                       > to 192.0.2.160 via ge-0/2/3.0, label-switched-path PE2_PE1
192.0.2.2/32          *[Direct/0] 00:34:58
                       > via lo0.1
192.51.100.1/32       *[OSPF/10] 00:34:12, metric 1
                       > to 192.51.100.51 via ge-0/2/1.0
192.51.100.2/32       *[BGP/170] 00:11:39, MED 1, localpref 100, from 192.0.2.1
                         AS path: 64513 I, validation-state: unverified
                       > to 192.0.2.160 via ge-0/2/3.0, label-switched-path PE2_PE1
192.51.100.50/31      *[Direct/0] 00:34:58
                       > via ge-0/2/1.0
192.51.100.50/32      *[Local/0] 00:34:58
                         Local via ge-0/2/1.0
192.51.100.64/31      *[OSPF/10] 00:16:11, metric 2
                       > to 192.51.100.51 via ge-0/2/1.0
203.0.113.0/24        *[BGP/170] 00:11:44, MED 0, localpref 100, from 192.0.2.1
                         AS path: 64513 I, validation-state: unverified
                       > to 192.0.2.160 via ge-0/2/3.0, label-switched-path PE2_PE1
224.0.0.5/32          *[OSPF/10] 00:34:58, metric 1
                         MultiRecv
```

You can see here that "vrf-target" handles route distribution via MP-BGP, which is used by data plane to forward traffic. Finally, let's check data plane reachability:

```
lab@VIENNA> ping 203.0.113.1
PING 203.0.113.1 (203.0.113.1): 56 data bytes
64 bytes from 203.0.113.1: icmp_seq=0 ttl=63 time=0.812 ms
64 bytes from 203.0.113.1: icmp_seq=1 ttl=63 time=0.821 ms
64 bytes from 203.0.113.1: icmp_seq=2 ttl=63 time=0.773 ms
64 bytes from 203.0.113.1: icmp_seq=3 ttl=63 time=0.757 ms
^C
--- 203.0.113.1 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.757/0.791/0.821/0.027 ms

lab@VIENNA> ping 192.51.100.2
PING 192.51.100.2 (192.51.100.2): 56 data bytes
64 bytes from 192.51.100.2: icmp_seq=0 ttl=63 time=0.835 ms
64 bytes from 192.51.100.2: icmp_seq=1 ttl=63 time=0.763 ms
^C
--- 192.51.100.2 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.763/0.799/0.835/0.036 ms
```

Solution 2: Using BGP Extended Community: Domain ID

A sham link can only be used in cases where both sites are placed in same OSPF area and domain, but when sites are placed in different areas you can make use of BGP extended communities like *domain ID* to exchange routes. It's 8-byte, which is considered a BGP extended community attribute. When MP-BGP redistributes VPNv4 routes to other PE routers, the domain ID is carried with the routes as an extended community. With domain ID community, the PE router redistributes routes learned from its MP-BGP to OSPF domain, checking domain ID to decide whether the routes should be redistributed as inter-area (same domain ID) routes or external (different domain ID) routes to the CE router, as shown in Figure 5.2. External routes learned from remote PEs are distributed as external routes back to CE routers though domain ID matches. When both sites are in the same OSPF area, with same domain ID the internal routes will be redistributed as Summary LSAs (LSA Type 3) instead of LSA Type 1 and LSA Type 2.

NOTE If no domain ID is present on the received route and no domain ID is on the local OSPF VRF instance, then the route is redistributed as LSA Type 3. Also, if there is a different domain ID on the received route and local OSPF VRF, then this leads to redistribution of all LSAs as External LSAs.

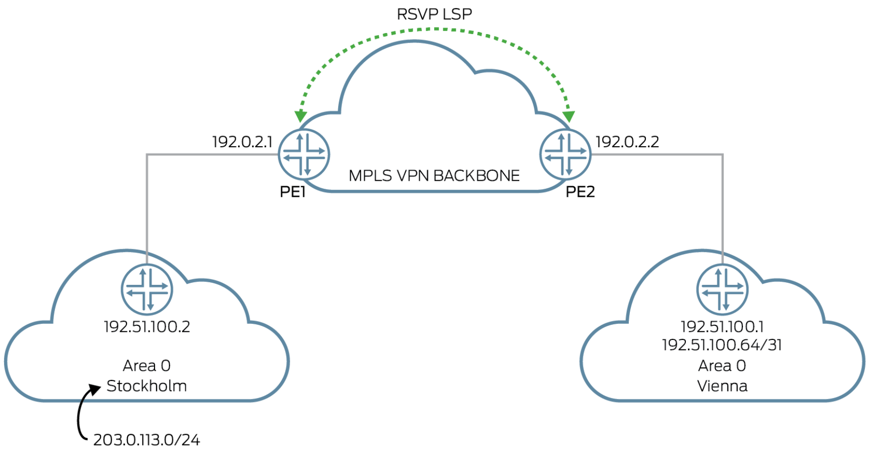


Figure 5.2 This Recipe's Domain ID Solution Topology

The first task is to get the basic topology set up, getting MP-BGP established between PE routers (PE1 and PE2), and getting the MPLS LSPs (using LDP or RSVP signaling) in place. RSVP LSP is used between PE1 and PE2, so let's start with the configuration of routing instance alias VRFs on both PE1 and PE2.

To configure Domain ID BGP extended community to extend OSPF routes:

Start with the configuration of the MX Series routers:

```
lab@STOCKHOLM> show configuration protocols ospf
area 0.0.0.0 {
    interface ge-0/2/2.0;     >> Interface connected to PE1
    interface lo0.0;
}
lab@VIENNA> show configuration protocols ospf
area 0.0.0.0 {
    interface ge-0/2/4.0;     >> Interface connected to PE2
    interface lo0.0;
    interface ge-0/2/5.0;     >> Interface connected to another OSPF speaker in VIENNA cloud
}
```

Now configure the PE routers:

```
lab@PE1> show configuration routing-instances STOCKHOLM_VRF
instance-type vrf;
interface ge-0/2/1.0;
interface lo0.1
route-distinguisher 192.0.2.1:64513;
vrf-import STOCKHOLM_VRF_IMPORT;
vrf-export STOCKHOLM_VRF_EXPORT;
protocols {
    ospf {
        domain-id 2.2.2.2;
        export EXPORT_OSPF_STOCKHOLM;
        area 0.0.0.0 {
            interface ge-0/2/1.0;
            interface lo0.1
        }
    }
}

lab@PE2> show configuration routing-instances VIENNA_VRF
instance-type vrf;
interface ge-0/2/1.0;
interface lo0.1
route-distinguisher 192.0.2.2:64512;
vrf-import VIENNA_VRF_IMPORT;
vrf-export VIENNA_VRF_EXPORT;
protocols {
    ospf {
        domain-id 2.2.2.2;
        export EXPORT_OSPF_VIENNA;
        area 0.0.0.0 {
            interface lo0.1
            interface ge-0/2/1.0;
        }
    }
}
```

The configuration uses the VRF import and export policies instead of using the `vrf-target` knob, which helps in the exchange of MP-BGP routes. (You can use `vrf-target` to achieve this, too.) Additionally, you need to export BGP routes to OSPF, which is achieved by policies `EXPORT_OSPF_VIENNA` and `EXPORT_OSPF_STOCKHOLM`.

CAUTION Use the same domain ID for all the sites so that all routes will not be exchanged as LSA Type 5 or LSA Type 7(NSSA), apart from those that are eligible.

Now configure the export and import policies mentioned earlier in this recipe:

```
lab@PE1> show configuration policy-options policy-statement EXPORT_OSPF_STOCKHOLM
term ACCEPT {
    from {
        protocol bgp;
        community STOCKHOLM_TARGET;
    }
    then accept;
}
term REJECT_ALL {
    then reject;
}
lab@PE1> show configuration policy-options policy-statement STOCKHOLM_VRF_IMPORT
term ACCEPT {
    from {
        protocol bgp;
        community STOCKHOLM_TARGET;
    }
    then accept;
}
term REJECT_ALL {
    then reject;
}
lab@PE1> show configuration policy-options policy-statement STOCKHOLM_VRF_EXPORT
term ACCEPT {
    from protocol ospf;
    then {
        community add STOCKHOLM_TARGET;
        community add DOMAIN;
        accept;
    }
}
term DIRECT {
    from protocol direct;
    then {
        community add STOCKHOLM_TARGET;
        accept;
    }
}
term REJECY_ALL {
    then reject;
}
lab@PE1> show configuration policy-options community STOCKHOLM_TARGET
members target:123:100;
lab@PE1> show configuration policy-options community DOMAIN
members domain-id:2.2.2.2:0;
lab@PE2> show configuration policy-options policy-statement EXPORT_OSPF_VIENNA
term ACCEPT {
    from {
        protocol bgp;
        community VIENNA_TARGET;
    }
    then accept;
}
term REJECT_ALL {
    then reject;
}
lab@PE2> show configuration policy-options policy-statement VIENNA_VRF_IMPORT
term ACCEPT {
    from {
        protocol bgp;
        community VIENNA_TARGET;
    }
    then accept;
}
term REJECT_ALL {
    then reject;
}
lab@PE2> show configuration policy-options policy-statement VIENNA_VRF_EXPORT
term ACCEPT {
    from protocol ospf;
    then {
        community add VIENNA_TARGET;
        community add DOMAIN;
        accept;
    }
}
term DIRECT {
    from protocol direct;
    then {
        community add VIENNA_TARGET;
        accept;
    }
}
term REJECY_ALL {
    then reject;
}
lab@PE2> show configuration policy-options community VIENNA_TARGET
members target:123:100;
lab@PE2> show configuration policy-options community DOMAIN
members domain-id:2.2.2.2:0;
```

Make sure the direct routes, i.e. interface and loopback routes of concerned VRFs, are exported to achieve data plane reachability.

To verify the working of the domain ID:

Let's verify the OSPF database of the MX Series routers:

```
lab@STOCKHOLM> show ospf database

    OSPF database, Area 0.0.0.0
 Type       ID               Adv Rtr           Seq      Age  Opt  Cksum  Len
Router   192.0.2.1        192.0.2.1        0x8000002a   241  0x22 0x5246  48
Router  *192.51.100.2     192.51.100.2     0x80000027   240  0x22 0x8d4a  48
Network *192.51.100.61    192.51.100.2     0x80000020  1171  0x22 0xa4f4  32
Summary  192.51.100.1     192.0.2.1        0x8000001d  1395  0xa2 0x6e14  28
Summary  192.51.100.64    192.0.2.1        0x8000001c  1995  0xa2 0xfb48  28
    OSPF AS SCOPE link state database
 Type       ID               Adv Rtr           Seq      Age  Opt  Cksum  Len
Extern   192.0.2.2        192.0.2.1        0x8000001c   795  0xa2 0x1d4   36
Extern   192.51.100.50    192.0.2.1        0x8000001c   195  0xa2 0x7898  36
Extern  *203.0.113.0      192.51.100.2     0x8000001e  2171  0x22 0x629b  36
lab@VIENNA> show ospf database

    OSPF database, Area 0.0.0.0
 Type       ID               Adv Rtr           Seq      Age  Opt  Cksum  Len
Router   192.0.2.2        192.0.2.2        0x80000026   328  0x22 0xa6ef  48
Router  *192.51.100.1     192.51.100.1     0x80000024   327  0x22 0x6f2b  60
Router   192.51.100.65    192.51.100.65    0x80000021   328  0x22 0x4468  36
Network *192.51.100.51    192.51.100.1     0x8000001f   327  0x22 0xa673  32
Network  192.51.100.65    192.51.100.65    0x8000001f   328  0x22 0x7284  32
Summary  192.51.100.2     192.0.2.2        0x8000001d   328  0xa2 0x5e22  28
    OSPF AS SCOPE link state database
 Type       ID               Adv Rtr           Seq      Age  Opt  Cksum  Len
Extern   192.51.100.60    192.0.2.2        0x8000001c   328  0xa2 0xf0dc  36
Extern   192.0.2.1        192.0.2.2        0x8000001c   328  0xa2 0x7998  36
Extern   203.0.113.0      192.0.2.2        0x8000001d   328  0xa2 0x286f  36
```

Observe that all the routes that were learned through OSPF, on each remote site, have changed their LSA types to Type 3 (Summary LSA) and that the 203.0.113.0 route retains its LSA Type 5 (External LSA). The routes that were local to the PE routers are, by default, exported as External LSAs.

Let's continue with the domain ID tag verification:

```
lab@PE1> show route table STOCKHOLM_VRF.inet.0 192.51.100.64 extensive | match Communities
                Communities: target:123:100 domain-id:2.2.2.2:0 rte-type:0.0.0.0:1:0
lab@PE1> show route table STOCKHOLM_VRF.inet.0 192.51.100.1 extensive | match Communities
                Communities: target:123:100 domain-id:2.2.2.2:0 rte-type:0.0.0.0:1:0
lab@PE1> show route table STOCKHOLM_VRF.inet.0 192.51.100.50 extensive | match Communities
                Communities: target:123:100
lab@PE2> show route table VIENNA_VRF.inet.0 192.51.100.2 extensive | match Communities
                Communities: target:123:100 domain-id:2.2.2.2:0 rte-type:0.0.0.0:1:0
lab@PE2> show route table VIENNA_VRF.inet.0 192.0.2.1 extensive | match Communities
                Communities: target:123:100
lab@PE2> show route table VIENNA_VRF.inet.0 203.0.113.0 extensive | match Communities
                Communities: target:123:100 domain-id:2.2.2.2:0 rte-type:0.0.0.0:5:1
```

You can verify that the routes are tagged with extended BGP community domain IDs. Moreover, the `rte-type` displays the area number, the LSA type, and options. The syntax of `rte-type` is `area number:LSA type:options`.

Now let's verify the route tables in the MX Series routers:

```
lab@STOCKHOLM> show route table inet.0

inet.0: 10 destinations, 10 routes (10 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

192.51.100.60/31   *[Direct/0] 02:43:03
                    > via ge-0/2/2.0
192.51.100.61/32   *[Local/0] 02:43:03
                      Local via ge-0/2/2.0
192.0.2.1/32       *[OSPF/10] 00:34:13, metric 1
                    > to 192.51.100.60 via ge-0/2/2.0
192.0.2.2/32       *[OSPF/150] 00:03:15, metric 0, tag 3489725929
                    > to 192.51.100.60 via ge-0/2/2.0
192.51.100.1/32    *[OSPF/10] 00:34:13, metric 2
                    > to 192.51.100.60 via ge-0/2/2.0
192.51.100.2/32    *[Direct/0] 02:43:04
                    > via lo0.0
192.51.100.50/31   *[OSPF/150] 00:03:15, metric 0, tag 3489725929
                    > to 192.51.100.60 via ge-0/2/2.0
192.51.100.64/31   *[OSPF/10] 00:34:13, metric 3
                    > to 192.51.100.60 via ge-0/2/2.0
203.0.113.0/24     *[Static/5] 02:07:37
```

```
                      Receive
224.0.0.5/32       *[OSPF/10] 03:15:02, metric 1
                      MultiRecv
lab@VIENNA> show route table inet.0

inet.0: 11 destinations, 11 routes (11 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

192.51.100.60/31   *[OSPF/150] 00:02:16, metric 0, tag 3489680734
                    > to 192.51.100.50 via ge-0/2/4.0
192.0.2.1/32       *[OSPF/150] 00:02:16, metric 0, tag 3489680734
                    > to 192.51.100.50 via ge-0/2/4.0
192.0.2.2/32       *[OSPF/10] 00:28:24, metric 1
                    > to 192.51.100.50 via ge-0/2/4.0
192.51.100.1/32    *[Direct/0] 02:30:32
                    > via lo0.0
192.51.100.2/32    *[OSPF/10] 00:28:18, metric 2
                    > to 192.51.100.50 via ge-0/2/4.0
192.51.100.50/31   *[Direct/0] 02:30:32
                    > via ge-0/2/4.0
192.51.100.51/32   *[Local/0] 02:30:32
                      Local via ge-0/2/4.0
192.51.100.64/31   *[Direct/0] 02:12:28
                    > via ge-0/2/5.0
192.51.100.64/32   *[Local/0] 02:12:28
                      Local via ge-0/2/5.0
203.0.113.0/24     *[OSPF/150] 00:28:18, metric 0, tag 3489680734
                    > to 192.51.100.50 via ge-0/2/4.0
224.0.0.5/32       *[OSPF/10] 03:14:17, metric 1
                      MultiRecv
```

Verify the route table of the PE VRFs:

```
lab@PE1> show route table STOCKHOLM_VRF.inet.0

STOCKHOLM_VRF.inet.0: 10 destinations, 10 routes (10 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

192.51.100.60/31   *[Direct/0] 02:43:03
                    > via ge-0/2/1.0
192.51.100.60/32   *[Local/0] 02:43:03
                      Local via ge-0/2/1.0
192.0.2.1/32       *[Direct/0] 02:43:03
                    > via lo0.1
192.0.2.2/32       *[BGP/170] 00:03:16, localpref 100, from 100.64.87.4
                      AS path: 64512 I
                    > to 192.0.2.129 via ge-0/2/0.0, label-switched-path PE1_PE2
192.51.100.1/32    *[BGP/170] 02:26:34, MED 1, localpref 100, from 100.64.87.4
                      AS path: 64512 I
                    > to 192.0.2.129 via ge-0/2/0.0, label-switched-path PE1_PE2
192.51.100.2/32    *[OSPF/10] 02:07:32, metric 1
                    > to 192.51.100.61 via ge-0/2/1.0
192.51.100.50/31   *[BGP/170] 00:03:16, localpref 100, from 100.64.87.4
                      AS path: 64512 I
                    > to 192.0.2.129 via ge-0/2/0.0, label-switched-path PE1_PE2
192.51.100.64/31   *[BGP/170] 02:12:04, MED 2, localpref 100, from 100.64.87.4
                      AS path: 64512 I
                    > to 192.0.2.129 via ge-0/2/0.0, label-switched-path PE1_PE2
203.0.113.0/24     *[OSPF/150] 02:07:37, metric 0, tag 0
                    > to 192.51.100.61 via ge-0/2/1.0
224.0.0.5/32       *[OSPF/10] 02:43:04, metric 1
                      MultiRecv
lab@PE2> show route table VIENNA
```

```
VIENNA_VRF.inet.0: 10 destinations, 10 routes (10 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

192.51.100.60/31   *[BGP/170] 00:02:16, localpref 100, from 192.0.2.1
                      AS path: 64513 I, validation-state: unverified
                    > to 192.0.2.160 via ge-0/2/3.0, label-switched-path PE2_PE1
192.0.2.1/32       *[BGP/170] 00:02:16, localpref 100, from 192.0.2.1
                      AS path: 64513 I, validation-state: unverified
                    > to 192.0.2.160 via ge-0/2/3.0, label-switched-path PE2_PE1
192.0.2.2/32       *[Direct/0] 02:30:32
                    > via lo0.1
192.51.100.1/32    *[OSPF/10] 02:29:46, metric 1
                    > to 192.51.100.51 via ge-0/2/1.0
192.51.100.2/32    *[BGP/170] 02:07:13, MED 1, localpref 100, from 192.0.2.1
                      AS path: 64513 I, validation-state: unverified
                    > to 192.0.2.160 via ge-0/2/3.0, label-switched-path PE2_PE1
192.51.100.50/31   *[Direct/0] 02:30:32
                    > via ge-0/2/1.0
192.51.100.50/32   *[Local/0] 02:30:32
                      Local via ge-0/2/1.0
192.51.100.64/31   *[OSPF/10] 02:11:45, metric 2
                    > to 192.51.100.51 via ge-0/2/1.0
203.0.113.0/24     *[BGP/170] 02:07:18, MED 0, localpref 100, from 192.0.2.1
                      AS path: 64513 I, validation-state: unverified
                    > to 192.0.2.160 via ge-0/2/3.0, label-switched-path PE2_PE1
224.0.0.5/32       *[OSPF/10] 02:30:32, metric 1
                      MultiRecv
```

Here, the MP-BGP learned routes for the remote sites OSPF routes, provides reachability to data plane traffic, since the PE router uses the MP-BGP routes to forward traffic to the remote sites.

And data plane reachability:

```
lab@STOCKHOLM> ping 192.51.100.65
PING 192.51.100.65 (192.51.100.65): 56 data bytes
64 bytes from 192.51.100.65: icmp_seq=0 ttl=62 time=0.835 ms
64 bytes from 192.51.100.65: icmp_seq=1 ttl=62 time=0.744 ms
^C
--- 192.51.100.65 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.744/0.789/0.835/0.046 ms

lab@VIENNA> ping 192.51.100.2
PING 192.51.100.2 (192.51.100.2): 56 data bytes
64 bytes from 192.51.100.2: icmp_seq=0 ttl=63 time=0.835 ms
64 bytes from 192.51.100.2: icmp_seq=1 ttl=63 time=0.763 ms
^C
--- 192.51.100.2 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.763/0.799/0.835/0.036 ms

lab@VIENNA> ping 203.0.113.1
PING 203.0.113.1 (203.0.113.1): 56 data bytes
64 bytes from 203.0.113.1: icmp_seq=0 ttl=63 time=0.812 ms
64 bytes from 203.0.113.1: icmp_seq=1 ttl=63 time=0.821 ms
64 bytes from 203.0.113.1: icmp_seq=2 ttl=63 time=0.773 ms
64 bytes from 203.0.113.1: icmp_seq=3 ttl=63 time=0.757 ms
^C
--- 203.0.113.1 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.757/0.791/0.821/0.027 ms
```

Discussion

When using a backdoor link between two sites (as shown in Figure 5.3) you can toggle the primary and backup role between the backdoor link and the MPLS VPN backbone.

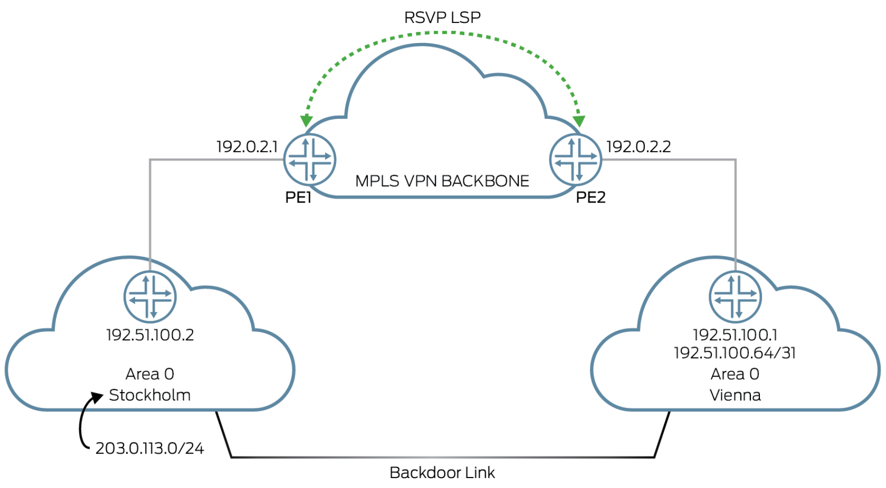


Figure 5.3 This Recipe's Topology – With Backdoor Link

When using a sham link you need to manipulate metric as in the following:

```
[edit]
lab@PE2# set routing-instances VIENNA_VRF protocols ospf area 0.0.0.0 sham-link-
remote 192.0.2.1 metric 10
```

When using domain ID to accomplish the OSPF route exchange, manipulate metric with the help of export policies configured under OSPF protocol for the respective routing-instance:

```
[edit]
lab@PE2# set policy-options policy-statement EXPORT_OSPF_
STOCKHOLM term ACCEPT from protocol bgp
lab@PE2# set policy-options policy-statement EXPORT_OSPF_
STOCKHOLM term ACCEPT from community STOCKHOLM_TARGET
lab@PE2# set policy-options policy-statement EXPORT_OSPF_
STOCKHOLM term ACCEPT then metric 10
lab@PE2# set policy-options policy-statement EXPORT_OSPF_STOCKHOLM term ACCEPT then accept
lab@PE2# set policy-options policy-statement EXPORT_OSPF_STOCKHOLM term REJECT_
ALL then reject
```

CAUTION When using a backdoor link with the MPLS VPN backbone exchanging OSPF routes, make sure to have a routing loop avoidance mechanism in place (example: route tag).

Recipe 6: Network Regression Testing with Junos PyEZ

You always want to make sure the functionality of your network does not degrade after upgrading software, committing configurations, or changing a physical topology. Even during normal operations there is always a need to check your work. This recipe shows you how to automate checks of your network's health using Junos PyEZ, a powerful Python library for Junos automation.

Problem

Administering networks requires a lot of repetitive tasks. Automating network administration, on the other hand, is a way to increase effectiveness, reduce downtime, and release human resources for more creative tasks.

Network automation comes in numerous flavors. You can automate configuration, monitoring, and event processing. You can write on-box scripts or orchestrate off-box, from a central server. Different possibilities lead to an initial question – where should you start automation efforts?

It makes sense to start with automating network monitoring, not only because it is very valuable, but because it allows you to feel the automation power while occupying a space in which it's safer to experiment (when compared to configuration automation).

Typically, to make sure your network is operating normally, you log in to several devices and check the command outputs (e.g., `show bgp summary`). Alternatively, you might look at graphs that represent different counters and values (such as CPU load, interface throughput, etc.). Why not make a program (or call it a script) that will do it for you? In this recipe, you will see how to do it with Junos PyEZ, a powerful yet simple library for automating Junos-based devices.

NOTE The scripts presented in this chapter are available on GitHub: https://github.com/pklimai/pyez-network-testing.

Solution

Software engineers use a common practice of writing "unit tests" – small pieces of code that test different parts of the main program (interestingly, sometimes tests are even written before the main code). Instead of the program, network engineers have their network – so let's write tests to be sure that our network is working fine!

Note that in different tests you will see several ways you can query run-time information from a Junos-based device:

- Directly processing XML output of Junos Remote Procedure Calls (RPCs)
- Using `jxmlease` library for easier processing of the XML output of RPC calls
- Using PyEZ built-in op Tables and Views
- Creating your own PyEZ op Table and View

Different methods have different advantages and use cases. After getting familiar with all of them, you can simply choose the ones that best suit your needs in a particular situation.

Demonstration Network and Prerequisites

You will use the network shown in Figure 6.1 as an example. It has a simple enterprise topology with three routers R1, R2, and R3. All routers are running OSPF in area 0.0.0.0, and IBGP is using lo0-based peering. Routers R1 and R2 have EBGP sessions attached to the ISPs and receive default routes from them.

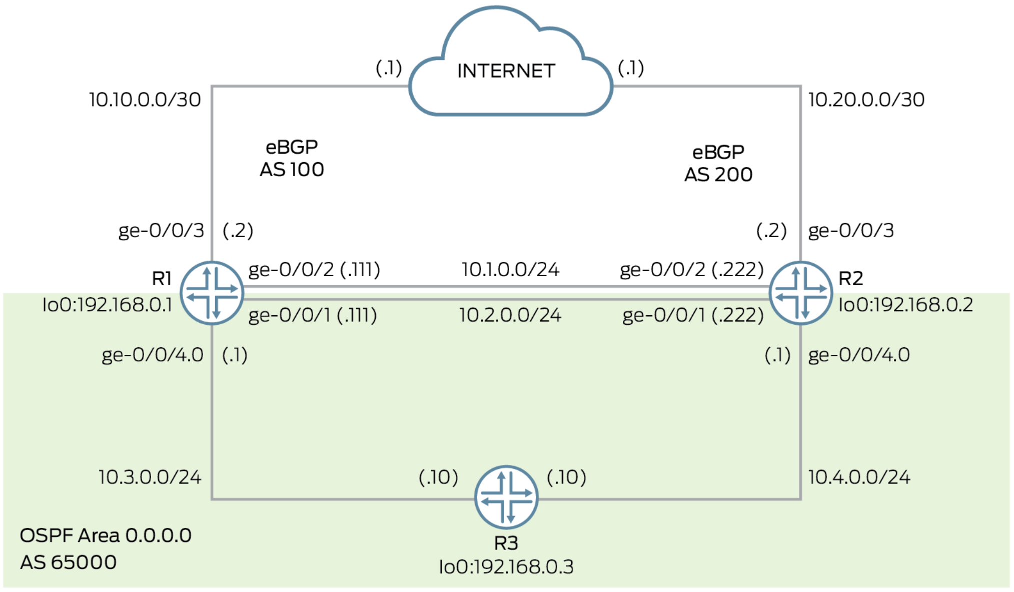


Figure 6.1 The Network Tested by PyEZ Scripts in this Recipe

In this recipe, you will write a set of tests to check OSPF, BGP, memory utilization, etc. Once you understand the examples you will be able to easily use similar testing methodology in your real network, even if the topology and the protocols used are significantly different.

For successful connections from PyEZ scripts, Netconf over SSH must be enabled on each Junos device, as shown in this example configuration:

```
[edit]
lab@R1# show system services
ssh;
netconf {
    ssh;
}
```

Python and PyEZ Installation

Next let's make sure the Python language interpreter is installed on your development host. For this recipe, the development host machine runs Ubuntu Linux and Python is pre-installed. To learn more about Python and its installation, visit https://www.python.org/.

For the steps required to install PyEZ on your operating system (just like Python, PyEZ is cross-platform) please refer to this Juniper technical publication: https://www.juniper.net/techpubs/en_US/junos-pyez2.0/topics/task/installation/junos-pyez-server-installing.html.

Starting from release 2.0, PyEZ supports both Python 2 and Python 3. The examples presented in this recipe were tested to work with Python 2.7 and Python 3.5, using PyEZ version 2.0.1.

The Base Script for the Testing Framework

The base script that you will use, `pyez-network-testing.py`, is presented below. It is a main file that calls all tests you will develop later. The actual tests must be placed in other files in the same directory, with names `tests_*.py`. The script's parts are numbered as **# (n)** for explanation:

```
from __future__ import print_function
from jnpr.junos import Device                              # (1)
from os.path import split, splitext, isfile, join
from os import listdir

script_dir = split(__file__)[0] or "."                     # (2)
for f in listdir(script_dir):                              # (3)
    if isfile(join(script_dir, f)) and f.startswith("tests_") and f.endswith(".py"):
        exec("from %s import *" % splitext(f)[0])

HOSTS = {                                                  # (4)
    "R1": "10.254.0.35",
    "R2": "10.254.0.37",
    "R3": "10.254.0.38",
}

USER = "lab"                                               # (5)
PASSWD = "lab123"

if __name__ == "__main__":                                 # (6)

    tests_success = 0                                      # (7)
    tests_fail = 0

    for host in sorted(HOSTS.keys()):                      # (8)
      print("Running tests for %s" % host)
      with Device(host=HOSTS[host], user=USER, passwd=PASSWD, gather_facts=False) as dev:#
(9)
         for name in dir():                                                     # (10)
            if name.startswith("test_%s_" % host) or name.startswith("test_all_"):    #
(11)
               print("    Running %s... " % name, end="")
               test_result = locals()[name](dev)  # (12)
               if test_result:                  # (13)
                  print(" pass")
                  tests_success += 1
               else:
                  print(" ***FAIL***")
                  tests_fail += 1

  print("--------")                          # (14)
  print("Network test script finished. Successful tests: %s, failed tests: %s" %
        (tests_success, tests_fail))
  print("All went OK." if tests_fail == 0 else "***WARNING***: There were failed tests!")
```

So, what is happening in this script?

1. You import the Device class from the PyEZ package, jnpr.junos. The instance of this class will handle a connection to a Junos device. Additionally, some functions needed to work with the file system are imported from the standard library. The import of `print_function` is used to make print() work the same way both in Python 3 and in Python 2.6+.

2. Get the directory in which script is located and save to `script_dir` variable.

3. For each file in `script_dir` whose name starts with `tests_` and ends with `.py`, import the contents of the file into the current namespace. Thus, all test functions get into the current module's namespace. This simple approach is adequate for our task.

4. `HOSTS` is a Python dictionary (associative array) with hostnames R1, R2, and R3 as keys. The corresponding values are management IP addresses. For a larger network, consider using a separate file or a database instead.

5. `USER` and `PASSWD` are, obviously, username and password on the managed devices and you hard code them in a script because this is a lab network. For production, it is highly recommended that you either ask that a password be entered from a keyboard every time, or use SSH keys instead.

6. Standard Python script "entry point".

7. Initialize two counters for successful and failed tests.

8. Loop for each of the `HOSTS` – in our case loop over R1, R2, R3.

9. Initialize PyEZ connection to the device and assign it to variable dev. Note the "context manager" syntax (`with` operator). In short, this means `open()` and `close()` functions will be called automatically when needed.

10. Loop over all names in the current scope (module). Remember, in Step 3, all tests were copied to the module's namespace, for loop will make the name variable run past all name values.

11. Check if name starts with `test_<hostname>_` or `test_all_`. If so, proceed.

12. Get reference to the function with name given by `name`, using `locals()[name]` construction (`locals()` returns a dictionary representing the current local symbol table). Call the function (particular network test), providing `dev` as a parameter. Save the result to `test_result` variable.

13. Note that `test_result` should be either `True` or `False`, meaning either a passed or failed test. Print the corresponding message and increase the proper counter.

14. When all tests on all devices are finished, print summary.

Okay, what's left now is to write the tests themselves.

Directly Processing XML Output of RPC Calls

The first option for writing tests you can employ is processing the XML response of a particular Junos XML RPC call. For example, let's check the chassis alarms (`show chassis alarms` is the corresponding CLI command). You can obtain the XML RPC from the CLI as follows:

```
lab@R1> show chassis alarms | display xml rpc
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/12.1X47/junos">
    <rpc>
        <get-alarm-information>
        </get-alarm-information>
    </rpc>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>
```

As you see, the RPC you need is `get-alarm-information`.

The corresponding output, in case of no alarms, is (both in clear text and XML form):

```
lab@R1> show chassis alarms
No alarms currently active

lab@R1> show chassis alarms | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/12.1X47/junos">
    <alarm-information xmlns="http://xml.juniper.net/junos/12.1X47/junos-alarm">
        <alarm-summary>
            <no-active-alarms/>
        </alarm-summary>
    </alarm-information>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>
```

As you see, if chassis alarms are absent, `alarm-summary/no-active-alarms` XML element is present (it might be not obvious but in the script we will be "rooted" at the `alarm-information` element and it should not be included in the XPath).

With this information, writing your first test is really easy (you can put it in the file named `tests_general.py`):

```
def test_all_chassis_alarms(dev):
    rpc_result = dev.rpc.get_alarm_information()
    return rpc_result.find("alarm-summary/no-active-alarms") is not None
```

This short function performs an RPC call to `dev` (note that hyphens must be replaced with underscores in the actual call to method `get_alarm_information()`). The result is saved to `rpc_result`, which will be of `lxml.etree._Element` type, allowing you to parse the XML with methods such as `find()`, `findall()`, `findtext()`, and `xpath()`.

MORE? More information on the `lxml` library and the mentioned methods can be found at the following URL: http://lxml.de/

The second line uses `find()` method to check if `alarm-summary/no-active-alarms` element exists. Function will return `True` (passed test) if it is actually present.

It is easy to write similar tests, for example to check system alarms and the existence of core files (dumps) on the device. Normally we would expect that neither alarms nor dumps are present; these tests can be put in the same file as `test_all_chassis_alarms`:

```
def test_all_system_alarms(dev):
    rpc_result = dev.rpc.get_system_alarm_information()
    return rpc_result.find("alarm-summary/no-active-alarms") is not None

def test_all_core_dumps(dev):
    rpc_result = dev.rpc.get_system_core_dumps()
    return rpc_result.find("directory/file-information") is None
```

A more complicated example that still uses the direct processing of XML RPC output is checking the existence of a BGP default route on R3. Normally, you expect the following state:

```
lab@R3> show route protocol bgp 0.0.0.0/0 exact

inet.0: 13 destinations, 14 routes (13 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

0.0.0.0/0          *[BGP/170] 21:23:26, localpref 100, from 192.168.0.1
                      AS path: 100 I
                    > to 10.3.0.1 via ge-0/0/3.0
                    [BGP/170] 21:23:26, localpref 100, from 192.168.0.2
                      AS path: 200 I
                    > to 10.4.0.1 via ge-0/0/4.0
```

This corresponds to the following XML RPC and XML response (output abbreviated):

```
lab@R3> show route protocol bgp 0.0.0.0/0 exact | display xml rpc
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/12.1X47/junos">
    <rpc>
        <get-route-information>
                <destination>0.0.0.0/0</destination>
                <exact/>
                <protocol>bgp</protocol>
        </get-route-information>
    </rpc>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>

lab@R3> show route protocol bgp 0.0.0.0/0 exact | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/12.1X47/junos">
    <route-information xmlns="http://xml.juniper.net/junos/12.1X47/junos-routing">
        <route-table>
            <table-name>inet.0</table-name>
...
            <rt junos:style="brief">
                <rt-destination>0.0.0.0/0</rt-destination>
                <rt-entry>
...
                    <learned-from>192.168.0.1</learned-from>
                    <as-path>100 I</as-path>
                    <nh>
                        <selected-next-hop/>
                        <to>10.3.0.1</to>
                        <via>ge-0/0/3.0</via>
                    </nh>
                </rt-entry>
                <rt-entry>
...
                    <learned-from>192.168.0.2</learned-from>
                    <as-path>200 I</as-path>
                    <nh>
                        <selected-next-hop/>
                        <to>10.4.0.1</to>
                        <via>ge-0/0/4.0</via>
                    </nh>
                </rt-entry>
            </rt>
        </route-table>
    </route-information>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>
```

The corresponding test that will check for the presence of two next hops for BGP 0.0.0.0/0 route on R3 might look like the following (you can put it in `tests_bgp.py` file):

```
def test_R3_bgp_default(dev):
    rpc_result = dev.rpc.get_route_information(protocol="bgp",destination="0.0.0.0/0",exac
t=True)
    nh_list = []
    for item in rpc_result.findall("route-table[table-name='inet.0']/rt[rt-
destination='0.0.0.0/0']/rt-entry/nh/to"):
        nh_list.append(item.findtext(".")) 
    nh_list.sort()
    return nh_list == ["10.3.0.1", "10.4.0.1"]
```

As you can see, the test's name starts with `test_R3_` which means it will run only on R3 by our main script. Note that a more complicated XPath expression in an argument of `findall()`allows you to extract all the BGP next hops (compare this XPath expression to the XML output above to see how it works; remember square brackets are used to denote the selection criteria).

And now it's time to run `pyez-network-testing.py` for the first time:

```
$ python pyez-network-testing.py
Running tests for R1
    Running test_all_chassis_alarms...  pass
    Running test_all_core_dumps...  pass
    Running test_all_system_alarms...  pass
Running tests for R2
    Running test_all_chassis_alarms...  pass
    Running test_all_core_dumps...  ***FAIL***
    Running test_all_system_alarms...  pass
Running tests for R3
    Running test_R3_bgp_default...  pass
    Running test_all_chassis_alarms...  pass
    Running test_all_core_dumps...  pass
    Running test_all_system_alarms...  pass
--------
Network test script finished. Successful tests: 9, failed tests: 1
***WARNING***: There were failed tests!
```

Wait, one test failed? `test_all_core_dumps` on R2? Let's see what happens on this device:

```
lab@R2> show system core-dumps
/var/crash/*core*: No such file or directory
-rw-rw----  1 root  wheel    3044927 Nov 24 22:32 /var/tmp/rpd.core.0.gz
/var/tmp/pics/*core*: No such file or directory
/var/crash/kernel.*: No such file or directory
/tftpboot/corefiles/*core*: No such file or directory
total files: 1
```

Really, a RPD core file exists on this router? After investigating the issue and removing the core file, let's run the script again:

```
$ python pyez-network-testing.py
Running tests for R1
    Running test_all_chassis_alarms...  pass
    Running test_all_core_dumps...  pass
    Running test_all_system_alarms...  pass
Running tests for R2
    Running test_all_chassis_alarms...  pass
    Running test_all_core_dumps...  pass
    Running test_all_system_alarms...  pass
Running tests for R3
    Running test_R3_bgp_default...  pass
    Running test_all_chassis_alarms...  pass
    Running test_all_core_dumps...  pass
    Running test_all_system_alarms...  pass
--------
Network test script finished. Successful tests: 10, failed tests: 0
All went OK.
```

Great! All the tests that you created have passed. Let's create some more.

Using jxmlease Library

Parsing XML with XPath expressions, as you've just done, is not always fun. A Python package called jxmlease (https://github.com/Juniper/jxmlease) can be used to *convert* XML to native Python data structures (lists, dictionaries, and their combinations) for easier access to XML elements.

NOTE To use jxmlease, you will need to install it. Consult https://github.com/Juniper/jxmleasehttps://github.com or just use the `pip install jxm-lease` command.

As an example, this XML:

```
lab@R1> show chassis routing-engine | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/12.1X47/junos">
    <route-engine-information xmlns="http://xml.juniper.net/junos/12.1X47/junos-chassis">
        <route-engine>
            <status>Testing</status>
            <memory-system-total>2048</memory-system-total>
            <memory-system-total-used>963</memory-system-total-used>
            <memory-system-total-util>47</memory-system-total-util>
            <memory-control-plane>1150</memory-control-plane>
            <memory-control-plane-used>334</memory-control-plane-used>
            <memory-control-plane-util>29</memory-control-plane-util>
...
```

...will be mapped to the following nested dictionaries:

```
{'route-engine-information':
    {'route-engine': {
         'status': 'Testing',
         'cpu-background': '0',
         'memory-system-total': '2048',
         'memory-system-total-used': '963',
         'memory-system-total-util': '47',
         'memory-control-plane': '1150',
         'memory-control-plane-used': '334',
         'memory-control-plane-util': '29',
...
```

... and you will be able to access the corresponding elements using dictionary indexing, as done in the following test (file `tests_memory.py`):

```
import jxmlease

def test_all_total_memory_percent_util(dev):
    parser = jxmlease.EtreeParser()
    res = parser(dev.rpc.get_route_engine_information())
    return int(res[“route-engine-information”][“route-engine”][“memory-system-total-
util”]) < 80
```

Here, the `parser` variable is an instance of the `EtreeParser` class defined in the jxmlease package. It is used to convert XML (in the form of `lxml.etree._Element`) to Pythonic data structure `res`. This data structure is then accessed using regular (but nested) indexing, `res[“route-engine-information”][“route-engine”][“memory-system-total-util”]`, to extract the required data (total memory utilization in this case).

Note that the 80% hard coded in this test as the threshold for total memory utilization is arbitrary and you might want to set a different value for your particular environment. Additionally, using a set of constants defined in some central place should be considered a better approach then hard coding the values inside the tests.

Another example test checks for the existence of a specific ARP record on R1’s uplink (file `tests_arp.py`):

```
import jxmlease

def test_R1_arp_gateway(dev):
    R1_GW_CORRECT_ARP_ENTRY = {
        ‘arp-table-entry-flags’: {‘none’: ‘‘},
        ‘hostname’: ‘10.10.0.1’,
        ‘interface-name’: ‘ge-0/0/3.0’,
        ‘ip-address’: ‘10.10.0.1’,
        ‘mac-address’: ‘00:0c:29:60:25:80’
    }
    parser = jxmlease.EtreeParser()
    res = parser(dev.rpc.get_arp_table_information())
    for item in res[‘arp-table-information’][‘arp-table-entry’]:
        if item == R1_GW_CORRECT_ARP_ENTRY:
            return True
    return False
```

Clearly, this test will fail if the MAC address of the gateway router changes, which might or might not be what you want. As usual, modify the test accordingly, to suit your network.

You will see the results of running all of the developed tests soon.

Using PyEZ Op Tables and Views

PyEZ uses a concept of op (operational) tables and views as another way to simplify interaction with RPC’s XML output. The idea is that particular fields from the XML document are mapped to specific fields of the Python data structure that is created automatically.

For example, this XML output:

```
lab@R1> show ospf neighbor extensive | display xml
<rpc-reply xmlns:junos=”http://xml.juniper.net/junos/12.1X47/junos”>
    <ospf-neighbor-information xmlns=”http://xml.juniper.net/junos/12.1X47/junos-routing”>
```

```
        <ospf-neighbor>
            <neighbor-address>10.1.0.222</neighbor-address>
            <interface-name>ge-0/0/1.0</interface-name>
            <ospf-neighbor-state>Full</ospf-neighbor-state>
            <neighbor-id>192.168.0.2</neighbor-id>
...
```

... can be collected to a table using the built-in `OspfNeighborTable` class to a standard PyEZ distribution. The corresponding test, that only cares about `ospf-neighbor-state` and the number of neighbors, is hopefully worth a thousand words (the file is called `tests_ospf.py`; note that in this example you create a function named `check_ospf_full_adjacencies()` and then reuse it in three different tests – reusing code is always a good idea):

```
from jnpr.junos.op.ospf import OspfNeighborTable

def check_ospf_full_adjacencies(dev, neighbor_count):
    ospf_table = OspfNeighborTable(dev)    # Create an instance of the Table
    ospf_table.get()                       # Populate the Table
    if len(ospf_table) != neighbor_count:
        return False
    for neighbor in ospf_table:
        if neighbor["ospf_neighbor_state"] != "Full":
            return False
    return True

def test_R1_ospf(dev):
    return check_ospf_full_adjacencies(dev, 3)

def test_R2_ospf(dev):
    return check_ospf_full_adjacencies(dev, 3)

def test_R3_ospf(dev):
    return check_ospf_full_adjacencies(dev, 2)
```

Here, `OspfNeighborTable` mapped `ospf-neighbor/ospf-neighbor-state` XML elements to dictionary entries with the key of "`ospf_neighbor_state`" – for each of the OSPF neighbors.

MORE? If you are interested, the definition of `OspfNeighborTable` can be found at GitHub: https://github.com/Juniper/py-junos-eznc/blob/master/lib/jnpr/junos/op/ospf.yml. You can browse the complete list of built-in tables/views at the following URL: https://github.com/Juniper/py-junos-eznc/tree/master/lib/jnpr/junos/op. You will see how tables/views are defined very shortly.

Creating a Custom Op Table and View

Although PyEZ has several dozen predefined tables/views, Junos has so many commands and corresponding RPCs that there is always a chance you will not find one you need for getting particular data. Do not get upset in such a situation, as creating custom Tables/Views is easy.

For example, `show bgp summary` command has no corresponding Table/View in the current PyEZ version. Let's create them. First, you look at the corresponding text and XML outputs, and obtain the XML RPC:

```
lab@R2> show bgp summary
Groups: 2 Peers: 3 Down peers: 0
Table          Tot Paths  Act Paths Suppressed    History Damp State    Pending
inet.0                 2          1          0          0          0          0
Peer                     AS      InPkt     OutPkt    OutQ   Flaps Last Up/Dwn State|#Active/
Received/Accepted/Damped...
10.20.0.1               200       5621       5641       0       2 1d 18:35:17 1/1/1/0    0/0/0/0
192.168.0.1           65000       6709       6709       0       0 2d 2:38:24  0/1/1/0    0/0/0/0
192.168.0.3           65000       6681       6705       0       2 1d 18:35:02 0/0/0/0    0/0/0/0

lab@R2> show bgp summary | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/12.1X47/junos">
    <bgp-information xmlns="http://xml.juniper.net/junos/12.1X47/junos-routing">
        <group-count>2</group-count>
        <peer-count>3</peer-count>
        <down-peer-count>0</down-peer-count>
        <bgp-rib junos:style="brief">
            <name>inet.0</name>
...
        </bgp-rib>
        <bgp-peer junos:style="terse" heading="Pe
er                     AS      InPkt     OutPkt    OutQ   Flaps Last Up/Dwn State|#Active/
Received/Accepted/Damped...">
            <peer-address>10.20.0.1</peer-address>
            <peer-as>200</peer-as>
            <input-messages>5621</input-messages>
            <output-messages>5641</output-messages>
            <route-queue-count>0</route-queue-count>
            <flap-count>2</flap-count>
            <elapsed-time junos:seconds="153320">1d 18:35:20</elapsed-time>
            <peer-state junos:format="1/1/1/0              0/0/0/0">Established</peer-
state>
            <bgp-rib>
                <name>inet.0</name>
                <active-prefix-count>1</active-prefix-count>
                <received-prefix-count>1</received-prefix-count>
                <accepted-prefix-count>1</accepted-prefix-count>
                <suppressed-prefix-count>0</suppressed-prefix-count>
            </bgp-rib>
        </bgp-peer>
        <bgp-peer junos:style="terse">
            <peer-address>192.168.0.1</peer-address>
            <peer-as>65000</peer-as>
            <input-messages>6709</input-messages>
            <output-messages>6709</output-messages>
            <route-queue-count>0</route-queue-count>
            <flap-count>0</flap-count>
            <elapsed-time junos:seconds="182307">2d 2:38:27</elapsed-time>
            <peer-state junos:format="0/1/1/0              0/0/0/0">Established</peer-
state>
...
</rpc-reply>

lab@R2> show bgp summary | display xml rpc
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/12.1X47/junos">
    <rpc>
```

```
        <get-bgp-summary-information>
        </get-bgp-summary-information>
    </rpc>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>
```

When creating a PyEZ table, you need to decide on the following main fields:

- `rpc` : an RPC that must be issued: in the present case, it will be `get-bgp-summary-information`;
- `item` : in this case, each table element will represent a BGP peer and correspond to XML element `bgp-peer`;
- `key`: field(s) that will be used as an unique key, in this case just `peer-address` is used, assuming different peers can't have the same IP;
- `view` : reference to a corresponding view, in this case `BgpSummaryView`.

Views in PyEZ are used to map the XML tag names in the RPC response to the property names used within the Python program. Overall, this works so that XML data appears as Python objects with certain attributes.

When creating `BgpSummaryView`, just choose the fields (XML elements) you need to obtain for each `bgp-peer`, and give them proper names. For example, you can map `peer-address` XML element to `peer_ip` field. Field names are, in fact, arbitrary.

The complete test that uses your custom table/view will look like this (file `tests_bgp_table.py`). This example only defines the test for R2:

```
from jnpr.junos.factory.factory_loader import FactoryLoader
import yaml

bgpYAML = """
---
BgpSummaryTable:
  rpc: get-bgp-summary-information
  item: bgp-peer
  key: peer-address
  view: BgpSummaryView

BgpSummaryView:
  fields:
    peer_ip: peer-address
    peer_as: peer-as
    peer_state: peer-state
"""

def test_R2_bgp_peers_established(dev):
    globals().update(FactoryLoader().load(yaml.load(bgpYAML)))
    bgp_table = BgpSummaryTable(dev).get()
    for peer in bgp_table:
        if peer[“peer_state”] != “Established”:
            return False
    return True
```

NOTE As you can see, tables and views are formatted using YAML. Its syntax is rather intuitive, but for more information please read about it at: https://en.wikipedia.org/wiki/YAML.

The above test uses “inline” table/view defined in a multi-line string `bgpYAML` directly in the Python file (you could use a separate file as well). Then it was loaded to global namespace and used to check if all configured BGP peers are in the Established state.

NOTE We have not covered all the options supported for Junos PyEZ op tables/views. For a complete discussion, please consult the Juniper TechLibrary at: https://www.juniper.net/techpubs/en_US/junos-pyez2.0/information-products/pathway-pages/junos-pyez-developer-guide.html.

Finally, let’s run all the tests again:

```
$ python pyez-network-testing.py
Running tests for R1
    Running test_R1_arp_gateway...  pass
    Running test_R1_bgp_default...  pass
    Running test_R1_ospf...  pass
    Running test_all_chassis_alarms...  pass
    Running test_all_core_dumps...  pass
    Running test_all_system_alarms...  pass
    Running test_all_total_memory_percent_util...  pass
Running tests for R2
    Running test_R2_arp_gateway...  pass
    Running test_R2_bgp_peers_established...  pass
    Running test_R2_ospf...  pass
    Running test_all_chassis_alarms...  pass
    Running test_all_core_dumps...  pass
    Running test_all_system_alarms...  pass
    Running test_all_total_memory_percent_util...  pass
Running tests for R3
    Running test_R3_bgp_default...  pass
    Running test_R3_ospf...  pass
    Running test_R3_ping_ISP1_GW...  pass
    Running test_R3_ping_ISP2_GW...  pass
    Running test_all_chassis_alarms...  pass
    Running test_all_core_dumps...  pass
    Running test_all_system_alarms...  pass
    Running test_all_total_memory_percent_util...  pass
--------
Network test script finished. Successful tests: 22, failed tests: 0
All went OK.
```

So far, so good!

Of course, you can schedule running of the testing scripts with, for example, the cron utility. And it is easy to make Python send you email with results - every time, or just in case some failures were encountered.

NOTE In the above output, some tests are present that were not detailed in this recipe, such as `test_R3_ping_ISP1_GW`. Complete code for all the tests as well as Junos-based device configurations used in this recipe are available at: https://github.com/pklimai/pyez-network-testing.

Discussion

This recipe provided a series of tests that can check the state of your network. Of course, when you perform network configuration changes, such as adding new devices and links, some old tests may fail just because they are unaware of the new "normal" state. You will need to modify the tests accordingly, and do it from time to time (alternatively – try to write tests so that they do not fail if "allowed" changes are made). But having an automated test suite will for sure outweigh the need to support a set of test scripts – as you've seen, writing them is easy once you get the idea.

The goal of this recipe was two-fold: get you familiar with Junos PyEZ and suggest a simple way network regression testing can be performed. Regardless of what exact tool you use for Junos automation, very soon you will start feeling its power. The capabilities are almost endless and your work rises to a completely new level.

MORE? For an example of an advanced testing framework, check out the JSNAPy tool available at GitHub under this URL: https://github.com/Juniper/jsnapy. An excellent *Day One* book covers JSNAPy here: http://www.juniper.net/us/en/training/jnbooks/day-one/automation-series/jsnapy/.

Recipe 7:
Selective Resource Sharing Across VPNs

MPLS VPNs can be used to isolate different customer networks from each other. In some cases, though, a service running in a VPN should be available to other VPNs as well. There can be several reasons as to why this is desirable:

- Customers want to exchanges services with each other
- A customer has multiple VPNs
- The operator has a service it wants to offer inside multiple customer VPNs

This cookbook recipe shows you how you can selectively *leak resources* (or subnets) between different VPNs.

Problem

There's a service running in VPN A but you need to make sure that this service is made available to VPN B and VPN C as well. Furthermore, you need to make sure that only specific hosts from the other VPNs are allowed access this resource.

Solution

The working topology for this cookbook recipe is as shown here.

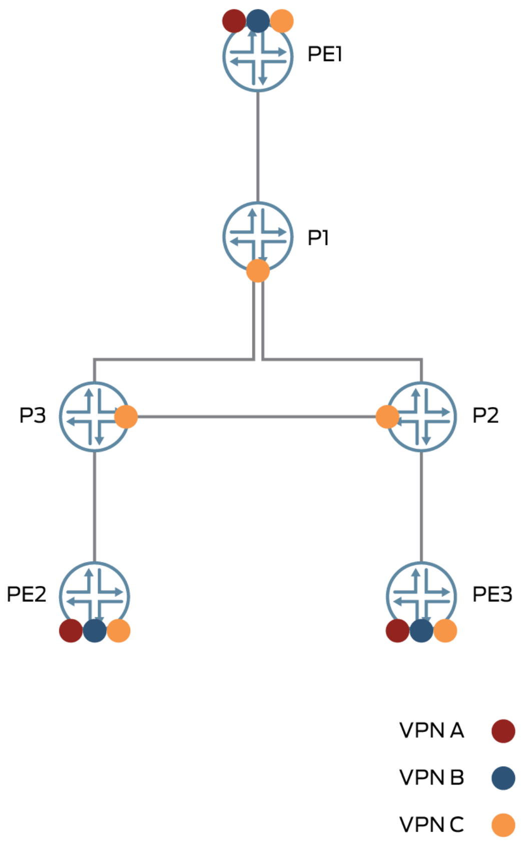


Figure 7.1 Topology for Selective Resource Sharing Across VPNs

Every site that is connected to a VPN will advertise a /24 route and a host route. You need to make sure that the host routes from VPN B and VPNC will be able to access a service in VPN A. This means that you need to realize IP connectivity between the following hosts:

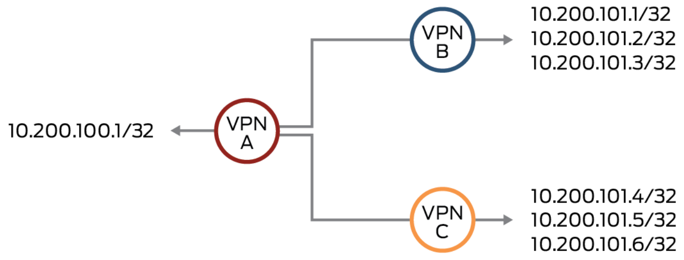


Figure 7.2 Host Connectivity

What you will not do is realize IP connectivity between any other host residing in any of the VPNs. You can do this by using different route targets and routing policies and by using the route targets:

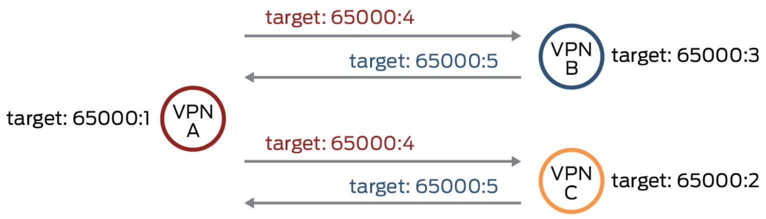


Figure 7.3 Route Targets

The black route targets are used for routes that should only be visible in their 'native' VPN. The red route-target is added to the service residing in VPNA. This is the service that we want to share with both VPN B and VPN C. The blue target is added to host routes that we want to make the service accessible to.

In Figure 7.3, route targets *target:65000:1*, *target:65000:2*, *target:65000:3* are used for routes that should only be visible in their 'native' VPN. Route target *target:65000:4* is added to the service residing in VPNA, the service that you want to share with both VPN B and VPN C. Route targets *target:65000:5* is added to host routes that you want to make the service accessible to.

Now configure the routing instances on all the PE routers in a similar way:

```
lab@PE1# show routing-instances
vpn-a {
    instance-type vrf;
    interface ge-0/0/2.100;
```

```
        vrf-import vpna-import;
        vrf-export vpna-export;
        vrf-table-label;
        protocols {
            bgp {
                group cpe {
                    family inet {
                        unicast;
                    }
                    peer-as 66000;
                    as-override;
                    neighbor 172.16.1.2;
                }
            }
        }
    }
    vpn-b {
        instance-type vrf;
        interface ge-0/0/2.200;
        vrf-import vpnb-import;
        vrf-export vpnb-export;
        vrf-table-label;
        protocols {
            bgp {
                group cpe {
                    family inet {
                        unicast;
                    }
                    peer-as 66001;
                    as-override;
                    neighbor 172.17.1.2;
                }
            }
        }
    }
    vpn-c {
        instance-type vrf;
        interface ge-0/0/2.300;
        vrf-import vpnc-import;
        vrf-export vpnc-export;
        vrf-table-label;
        protocols {
            bgp {
                group cpe {
                    family inet {
                        unicast;
                    }
                    peer-as 66001;
                    as-override;
                    neighbor 172.18.1.2;
                }
            }
        }
    }
```

The policies used for all three VPNs are the same as well. The following is the example policy for VPN A:

```
lab@PE2# show policy-options
policy-statement vpna-export {
    term 1 {
        from protocol [ bgp direct ];
        then {
            community add vpna;
            accept;
        }
    }
}
policy-statement vpna-import {
    term 1 {
        from {
            protocol bgp;
            community vpna;
        }
        then accept;
    }
}
community vpna members target:65000:1;
```

So far, there should be nothing unusual for you. This is your standard no frills MPLS L3VPN with full mesh connectivity across your MPLS network. Configured like this, the different VPNs are nicely separated.

But let's now move to PE1. The first step is to alter the policy that you are using for VPN A. You need to alter the import and the export policy. What you need to do is add a shared-resource community to the host route giving access to the resource (10.200.100.1/32). The other thing you need to do is import the clients that need granted access to this shared resource. To accomplish this, configure the following:

```
[edit]
lab@PE1# set policy-options policy-statement vpna-export term 0 from route-
filter 10.200.100.1/32 exact

[edit]
lab@PE1# set policy-options policy-statement vpna-export term 0 then community add vpna

[edit]
lab@PE1# set policy-options policy-statement vpna-export term 0 then community add shared-
resource

[edit]
lab@PE1# set policy-options policy-statement vpna-export term 0 then accept

[edit]
lab@PE1# insert policy-options policy-statement vpna-export term 0 before term 1

[edit]
lab@PE1# set policy-options policy-statement vpna-import term 2 from protocol bgp

[edit]
lab@PE1# set policy-options policy-statement vpna-import term 2 from community clients

[edit]
lab@PE1# set policy-options policy-statement vpna-import term 2 then accept
```

```
[edit]
lab@PE1# set policy-options community clients members target:65000:5

[edit]
lab@PE1# set policy-options community shared-resource members target:65000:4
```

You have now instructed this VPN to advertise the 10.200.100.1 route with an additional community. The import policy for VPN A was also altered to import the client routes.

Now you need to configure the routing policies for VPN B and VPN C. As an example, let's look at the changes to VPN B:

```
[edit]
lab@PE1# rename policy-options policy-statement vpnb-import term 1 to term 2

[edit]
lab@PE1# set policy-options policy-statement vpnb-import term 1 from protocol bgp

[edit]
lab@PE1# set policy-options policy-statement vpnb-import term 1 from community shared-
resource

[edit]
lab@PE1# set policy-options policy-statement vpnb-import term 1 from route-
filter 10.200.100.1/32 exact

[edit]
lab@PE1# set policy-options policy-statement vpnb-import term 1 then accept

[edit]
lab@PE1# insert policy-options policy-statement vpnb-import term 1 before term 2

[edit]
```

This will make sure that the shared resource route will be imported into VPN B. Next, move on to the export policy:

```
[edit]
lab@PE1# rename policy-options policy-statement vpnb-export term 1 to term 2

[edit]
lab@PE1# set policy-options policy-statement vpnb-export term 1 from protocol bgp

[edit]
lab@PE1# set policy-options policy-statement vpnb-export term 1 from route-
filter 10.200.101.0/24 prefix-length-range /32-/32

[edit]
lab@PE1# set policy-options policy-statement vpnb-export term 1 then community add vpnb

[edit]
lab@PE1# set policy-options policy-statement vpnb-export term 1 then community add clients

[edit]
lab@PE1# set policy-options policy-statement vpnb-export term 1 then accept
```

```
[edit]
lab@PE1# insert policy-options policy-statement vpnb-export term 1 before term 2

[edit]
```

And this policy will make sure that the host routes within the 10.200.101.0/24 range will be exported with both the VPN route target as well as the shared resource route target.

There is one last, little catch. Altering the import and export policies for VPN B and VPN C on all PE routers will ensure that the routes are being shared on all PE routers *except* for the router that is local to the shared resource.

This is because the routing information is evaluated when received by a BGP peer. So PE2 and PE3 received the routes from PE1. The received routes are evaluated by the VPNs routing policy. After this, the routes are shared or 'leaked' between the VPNs. However, the PE1 router does not advertise this information to itself. In order to make PE1 evaluate the configured route policies for the locally configured routing instances, you need to add the `auto-export` knob to all the VPNs involved and that means adding it to VPN A, VPN B, and VPN C.

```
[edit]
lab@PE1# set routing-instances vpn-a routing-options auto-export

[edit]
lab@PE1# set routing-instances vpn-b routing-options auto-export

[edit]
lab@PE1# set routing-instances vpn-c routing-options auto-export
```

You can verify the configuration by issuing the following commands on any of the PE routers:

```
lab@PE1> show route community-name clients terse
...

vpn-a.inet.0: 14 destinations, 14 routes (14 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

A V Destination        P Prf   Metric 1   Metric 2  Next hop        AS path
* ? 10.200.101.1/32    B 170        100                              66001 I
  unverified                                       >172.17.1.2
* ? 10.200.101.2/32    B 170        100                              66001 I
  unverified                                       >10.0.0.6
* ? 10.200.101.3/32    B 170        100                              66001 I
  unverified                                       >10.0.0.6
* ? 10.200.101.4/32    B 170        100                              66001 I
  unverified                                       >172.18.1.2
* ? 10.200.101.5/32    B 170        100                              66001 I
  unverified                                       >10.0.0.6
* ? 10.200.101.6/32    B 170        100                              66001 I
  unverified                                       >10.0.0.6
```

```
vpn-b.inet.0: 11 destinations, 11 routes (11 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

A V Destination        P Prf   Metric 1   Metric 2  Next hop        AS path
* ? 10.200.101.2/32    B 170        100                             66001 I
  unverified                                       >10.0.0.6
* ? 10.200.101.3/32    B 170        100                             66001 I
  unverified                                       >10.0.0.6

vpn-c.inet.0: 11 destinations, 11 routes (11 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

A V Destination        P Prf   Metric 1   Metric 2  Next hop        AS path
* ? 10.200.101.5/32    B 170        100                             66001 I
  unverified                                       >10.0.0.6
* ? 10.200.101.6/32    B 170        100                             66001 I
  unverified                                       >10.0.0.6


lab@PE3> show route community-name shared-resource terse
...

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

A V Destination        P Prf   Metric 1   Metric 2  Next hop        AS path
* ? 10.200.100.1/32    B 170        100                             66000 I
  unverified                                       >10.0.0.10

vpn-b.inet.0: 11 destinations, 11 routes (11 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

A V Destination        P Prf   Metric 1   Metric 2  Next hop        AS path
* ? 10.200.100.1/32    B 170        100                             66000 I
  unverified                                       >10.0.0.10

vpn-c.inet.0: 11 destinations, 11 routes (11 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

A V Destination        P Prf   Metric 1   Metric 2  Next hop        AS path
* ? 10.200.100.1/32    B 170        100                             66000 I
  unverified                                       >10.0.0.10
```

Using this recipe's methodology, you can selectively share resources across VPNs and by altering the VRF routing policies you can make sure that a service is available to multiple VPNs. Cool.

Recipe 8:
Integrate MX Series Routers into Arbor Networks

The Arbor Networks SP product provides a reporting and visibility tool to detect DDoS attacks, profile network traffic, and monitor network path utilization. In order to provide these reports the product relies on getting information from the managed routers using the following protocols to collect data:

- SNMP polling for interface information
- Jflow for sampled information on all interfaces
- BGP peering to obtain route table data and monitor table changes

The SP appliance or virtual machine (VM) requires access to the MX Series for all three protocols to collect and store this information. Once the information is collected the system provides multiple reporting and alerting options based on the data.

Problem

Typically, Arbor Networks SP is deployed in a service provider or large enterprise network. The monitored routers need to be at the edge where Internet traffic ingresses to the network, providing the first point of access to detect anomalies in traffic and the best opportunity to deal with malicious traffic entering the network.

When the network has multiple upstream Internet peers, all MX Series routers involved in the various peerings need to be included in the monitoring solution to insure that all possible sources are visible for DDoS evaluations. And for traffic and transit management, all points of Internet ingress and egress are needed.

Integrating the SP device into the network to collect all three types of data from all the MX Series routers can be tricky and should be first done in your lab before being used in production environments.

Solution

Arbor Networks SP is deployed into the network using two network interfaces: one provides the SNMP, BGP, and web interface address, and the second interface is used for flow collection. The MX Series uses the loopback address for the BGP peer and SNMP collection and the fxp0 management address for the flow.

Table 8.1 Table IP Address Assignments

Device	Interface	Address	Protocols
Arbor SP	eth0	192.0.2.11/24	BGP; SNMP
	eth1	172.16.0.11/24	jflow
MX Router	lo0.0	127.0.0.1/32	BGP; SNMP
	fxp0.0	172.16.0.10/24	jflow

MX Series Configuration

The MX Series configuration consists of three parts: the SNMP, BGP, and Jflow configurations.

For SNMP you simply need to add the Arbor SP IP address as an allowed host for the SNMP community on the MX Series. This is used to pull interface statistics from the MX to the Arbor SP appliance. The `String` value is the SNMP community string that Arbor SP uses when collecting data from the MX:

```
[edit]
root@# set snmp community String clients 192.0.2.11/32
root@# set snmp community String authorization read-only
```

The next configuration has an iBGP to the MX loopback address from the Arbor SP eth0 interface. These peerings are set up as standard iBGP internal peers using the local AS with the default loopback addresses. The idea is to integrate the Arbor appliances into the upstream peerings:

```
[edit]
root@# set protocols bgp group Arbor type internal
root@# set protocols bgp group Arbor family inet unicast
root@# set protocols bgp group Arbor peer-as 64496
root@# set protocols bgp group Arbor neighbor 192.0.2.11 local-address 127.0.0.1
```

The Jflow configuration sends the flow data at a 1000 sample rate from the fxp0 interface to the dedicated flow collection interface eth1 on the Arbor SP. Also note the requirement that all the interfaces be configured to send flow data. Here a group configuration to automatically apply the jflow to all interfaces on the MX is used:

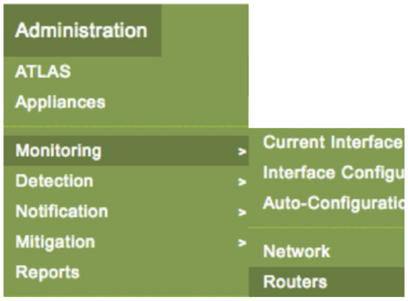
```
[edit]
root@# set forwarding-options sampling input rate 1000
root@# set forwarding-options sampling family inet output flow-
server 172.16.0.11 port 2055
root@# set forwarding-options sampling family inet output flow-server 172.16.0.11 source-
address 172.16.0.10
root@# set forwarding-options sampling family inet output flow-
server 172.16.0.11 version 5

root@# set groups jflow_subinterfaces interfaces <*-*> unit <*> family inet sampling input
root@# set groups jflow_subinterfaces interfaces <*-
*> unit <*> family inet sampling output
root@# set apply-groups jflow_subinterfaces
```

NOTE If your MX Series is using a "protect-re" firewall filter on the loopback interface, you will need to add the Arbor SP addresses to the sections that restrict SNMP and BGP traffic.

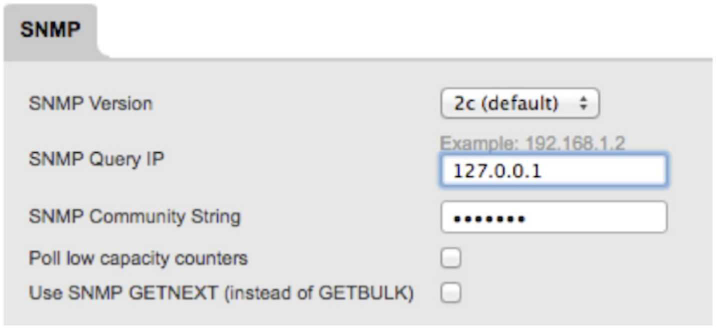
NOTE NTP must be configured and enabled for Jflow to work properly in the Junos OS.

Add MX Series to Arbor SP

To add the MX Series as a managed router, use the Arbor SP installation GUI interface, and follow this GUI path: Administration > Monitoring > Routers > Add Router button.



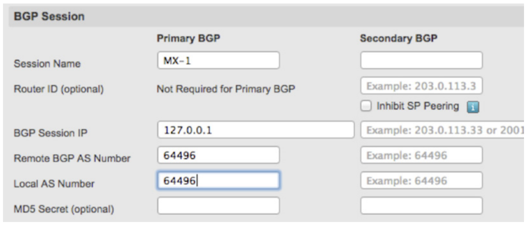
Use SNMP to add the address of the MX Series to poll and the community string and version to use.



SNMP

SNMP Version	2c (default)
SNMP Query IP	Example: 192.168.1.2 / 127.0.0.1
SNMP Community String	•••••••
Poll low capacity counters	☐
Use SNMP GETNEXT (instead of GETBULK)	☐

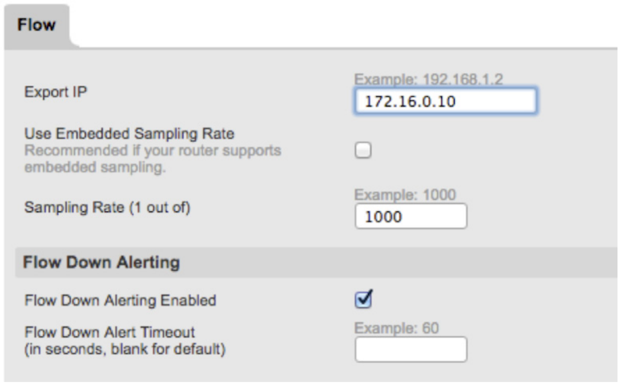
Use BGP to set up the peer address and the ASN.



BGP Session

	Primary BGP	Secondary BGP
Session Name	MX-1	
Router ID (optional)	Not Required for Primary BGP	Example: 203.0.113.3 / ☐ Inhibit SP Peering
BGP Session IP	127.0.0.1	Example: 203.0.113.33 or 2001
Remote BGP AS Number	64496	Example: 64496
Local AS Number	64496	Example: 64496
MD5 Secret (optional)		

And use the Flow settings to verify the address sending the flow data and the sampling rate for calculations.



Flow

Export IP	Example: 192.168.1.2 / 172.16.0.10
Use Embedded Sampling Rate Recommended if your router supports embedded sampling.	☐
Sampling Rate (1 out of)	Example: 1000 / 1000

Flow Down Alerting

Flow Down Alerting Enabled	✓
Flow Down Alert Timeout (in seconds, blank for default)	Example: 60

Verifications

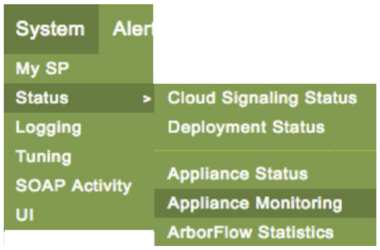
On the MX Series, SNMP get statistics should be incrementing. Run the `show snmp command` statistics several times and confirm the incrementing get responses:

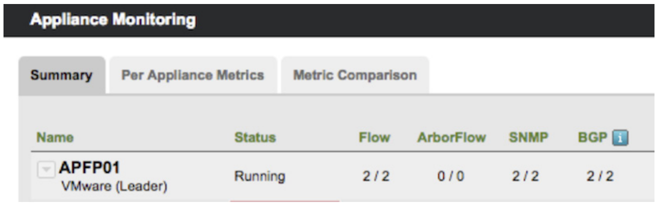
```
root@> show snmp statistics
SNMP statistics:
…
  Output:
    Packets: 18643, Too bigs: 0, No such names: 0,
```

Look in the message log for the IP address value of the Arbor SP appliance. The following sample log message shows an *unauthorized* SNMP query failure as an example. When all is working correctly no log messages are returned:

```
root@> show log messages | match 192.0.2.11
Jan  2 00:25:33  MX snmpd[4035]: SNMPD_AUTH_FAILURE: nsa_log_
community: unauthorized SNMP community from 192.0.2.11 to 127.0.0.1 (String)
    Get responses: 142392142392, Traps: 850995

Confirm the BGP peer is up and established on the MX:
root > show bgp neighbor 192.0.2.11
Peer: 193.0.2.11+3038 AS 33154 Local: 127.0.0.1+179 AS 64496
  Type: Internal    State: Established    Flags: <Sync RSync>
  Last State: EstabSync     Last Event: RecvKeepAlive
  Last Error: Hold Timer Expired Error
  Options: <Preference LocalAddress PeerAS Refresh>
  Local Address: 127.0.0.1 Holdtime: 90 Preference: 170
  Number of flaps: 14
  Last flap event: Closed
  Error: ‘Hold Timer Expired Error’ Sent: 4 Recv: 1
  Peer ID: 192.0.2.11     Local ID: 127.0.0.1      Active Holdtime: 90
  Keepalive Interval: 30         Group index: 16   Peer index: 0
  BFD: disabled, down
  NLRI for restart configured on peer: inet-unicast
  NLRI advertised by peer: inet-unicast inet-flow
  NLRI for this session: inet-unicast
  Peer does not support Refresh capability
  Stale routes from peer are kept for: 300
  Peer does not support Restarter functionality
  Peer does not support Receiver functionality
  Peer supports 4 byte AS extension (peer-as 33154)
  Peer does not support Addpath
  Table inet.0 Bit: 20007
    RIB State: BGP restart is complete
    Send state: in sync
    Active prefixes:              0
    Received prefixes:            0
    Accepted prefixes:            0
    Suppressed due to damping:    0
    Advertised prefixes:          620403
  Last traffic (seconds): Received 20   Sent 2    Checked 50
  Input messages:  Total 11289  Updates 0 Refreshes 0 Octets 214521
  Output messages: Total 906633 Updates 895002  Refreshes 0 Octets 105942039
  Output Queue[1]: 0
  Trace options: state
  Trace file: /var/log/bgp.log size 10485760 files 10
```

You can only verify the Jflow configuration on the MX although there isn't a mechanism to verify the flow data exporting to Arbor SP. But on the Arbor SP GUI interface you can confirm that the data for all three protocols is being received and recorded under SystemSystem > Appliance Monitoring.





Name	Status	Flow	ArborFlow	SNMP	BGP
APFP01 VMware (Leader)	Running	2 / 2	0 / 0	2 / 2	2 / 2

The numbers on each column confirm the number of routers sending data versus the number of routers configured for data collection. When all is working the numbers should be the same. The ArborFlow column is null because the two MX Series are using Jflow.

Recipe 9:
BGP Flow Spec Between Arbor Networks and MX Series

After the Arbor SP reporting and monitoring is configured for an MX Series router, there is an option to use BGP flow specification (flowspec) for mitigation announcements. For DDoS mitigation, the SP appliance can send BGP flow spec updates to compatible routers to automatically apply filters that either rate-limit or drop traffic per the filter specifications.

Problem

When a DDoS attack is in progress one potential avenue of mitigation is to drop traffic matching a specific criteria of the protocol, the port, and the destination IP address in particular combinations. These mitigations can occur by applying firewall filters to the appropriate interfaces on the MX Series router to discard the attack traffic.

BGP flowspec is a mitigation technique specified in RFC 5575 that allows such filters to be defined in a BGP peer advertisement and then applied to router interfaces. This technique can help automate the application of filters.

But the operator still needs to identify the properties of the malicious traffic in order to create the correct discard criteria for the filter. Then the operator needs to create the necessary mitigation filter for the attack traffic.

Solution

Arbor SP provides the operator with the alert information needed to identify the proper elements of the filter and a mechanism to send the BGP flowspec announcements to the MX Series router. After the MX router is set up for basic monitoring with Arbor SP, the BGP flowspec configuration can be added to the setup. This recipe provides the basics of that configuration for the MX Series..

The BGP flowspec advertisement can use BGP communities as criteria for import policies just as any other route advertisement. So the MX configuration adds a community to the advertisement to facilitate the handling of the updates. Table 9.1 lists the details of the address assignments.

Table 9.1 Table IP Address Assignments

Device	Interface	Address	Protocols
Arbor SP	eth0	192.0.2.11/24	BGP; SNMP
	eth1	172.16.0.11/24	jflow
MX Series Router	lo0.0	127.0.0.1/32	BGP; SNMP
	fxp0.0	172.16.0.10/24	jflow

MX Series Configuration

The MX Series configuration requires enabling BGP flowspec and then setting it up as the peer to the Arbor SP system to receive the announcements. The following enables the basic processing order:

```
[edit]
root@# set routing-options flow term-order standard
root@# set policy-options community SP_flowspec members 64496:500
root@# set policy-options policy-statement flowspec_import term flowspec_
in from community SP_flowspec
root@# set policy-options policy-statement flowspec_import term flowspec_in then accept
```

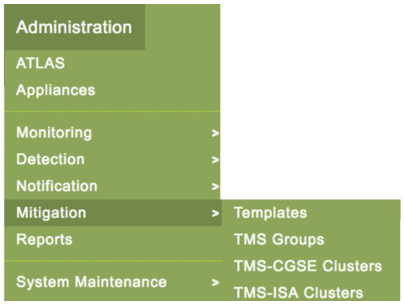
The next configuration creates an iBGP to the MX loopback address from the Arbor SP eth0 interface, adding the flow options to the peering and the flow import policy:

```
[edit]
root@# set protocols bgp group Arbor type internal
root@# set protocols bgp group APF family inet unicast
root@# set protocols bgp group Arbor peer-as 64496
root@# set protocols bgp group Arbor neighbor 192.0.2.11 local-address 127.0.0.1
root@# set protocols bgp group APF family inet flow
root@# set protocols bgp group APF family inet flow no-validate flowspec_import
```

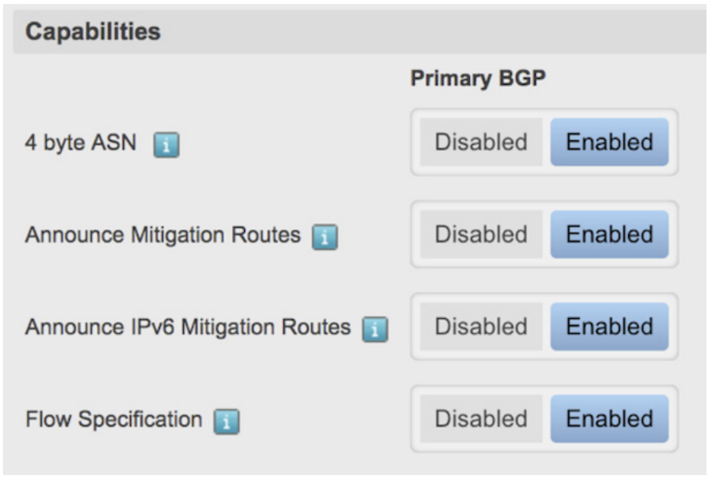
NOTE If your MX Series is using a “protect-re” firewall filter on the loopback interface, you will need to add the Arbor SP addresses to the sections that restrict BGP traffic.

Arbor SP: Add BGP Flow Spec to MX

You will need to add the MX Series as a managed router on the Arbor SP installation GUI, under Administration > Monitoring > Routers > .



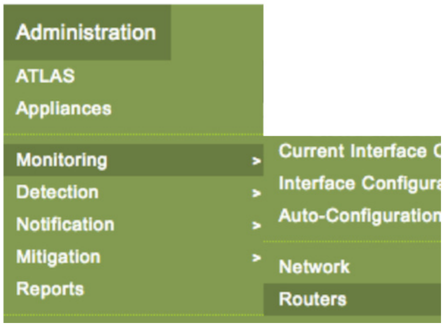
Edit the setup information for the MX Series that will receive the BGP flowspec announcement by selecting the BGP tab and enabling Flow Specification for the BGP peer.

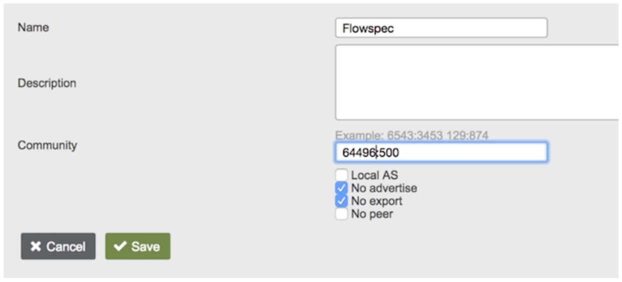


Capabilities

	Primary BGP
4 byte ASN	Disabled [✓ Enabled]
Announce Mitigation Routes	Disabled [✓ Enabled]
Announce IPv6 Mitigation Routes	Disabled [✓ Enabled]
Flow Specification	Disabled [✓ Enabled]

As an option, you can set up your BGP communities in advance so they can be selected for use during the BGP process. You can also enter the community values manually, later, but setting them up in advance prevents typos from occurring during the mitigation process. Go to: Administration > Mitigation > Community Groups > Create New.





Name: Flowspec

Description:

Community: 64496:500

Example: 6543:3453 129:874

[ ] Local AS
[✓] No advertise
[✓] No export
[ ] No peer

Create the community value and add the No advertise and No export options. Click Save.

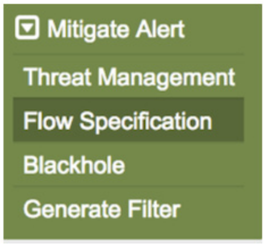
Verifications

On the MX Series, the BGP peer should now show that "address families" has inet-flow is enabled. Let's check the lab MX:

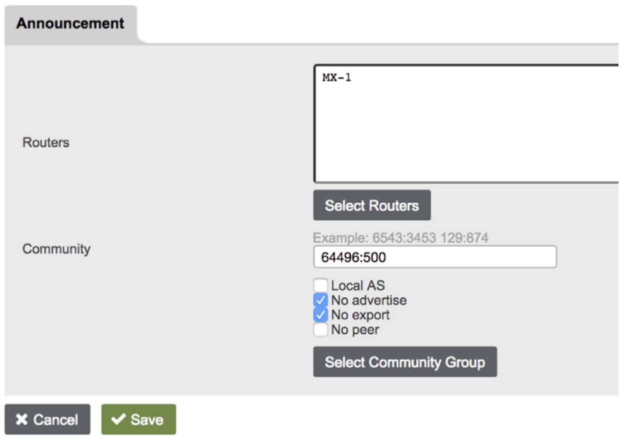
```
root > show bgp neighbor 192.0.2.11
Peer: 193.0.2.11+3038 AS 33154 Local: 127.0.0.1+179 AS 64496
  Type: Internal    State: Established    Flags: <Sync RSync>
  Last State: EstabSync     Last Event: RecvKeepAlive
  Last Error: Hold Timer Expired Error
  Options: <Preference LocalAddress PeerAS Refresh>
  Address families configured: inet-unicast inet-flow
  Local Address: 127.0.0.1 Holdtime: 90 Preference: 170
  Number of flaps: 14
  Last flap event: Closed
  Error: 'Hold Timer Expired Error' Sent: 4 Recv: 1
  Peer ID: 192.0.2.11     Local ID: 127.0.0.1      Active Holdtime: 90
  Keepalive Interval: 30         Group index: 16   Peer index: 0
  BFD: disabled, down
```

```
NLRI for restart configured on peer: inet-unicast
NLRI advertised by peer: inet-unicast inet-flow
NLRI for this session: inet-unicast
Peer does not support Refresh capability
Stale routes from peer are kept for: 300
Peer does not support Restarter functionality
Peer does not support Receiver functionality
Peer supports 4 byte AS extension (peer-as 33154)
Peer does not support Addpath
Table inet.0 Bit: 20007
  RIB State: BGP restart is complete
  Send state: in sync
  Active prefixes:              0
  Received prefixes:            0
  Accepted prefixes:            0
  Suppressed due to damping:    0
  Advertised prefixes:          620403
Last traffic (seconds): Received 20   Sent 2    Checked 50
Input messages:  Total 11289  Updates 0 Refreshes 0 Octets 214521
Output messages: Total 906633 Updates 895002  Refreshes 0 Octets 105942039
Output Queue[1]: 0
Trace options: state
Trace file: /var/log/bgp.log size 10485760 files 10
```

Okay, all is well. To send a BGP flowspec announcement go back to the Arbor SP GUI and choose Mitigate Alert from the DoS alert that you are working on, first making notes on the traffic tab of the details you will need for the filter, namely the destination address and the protocol and source port of the attack type (i.e. NTP reflection UDP 123).



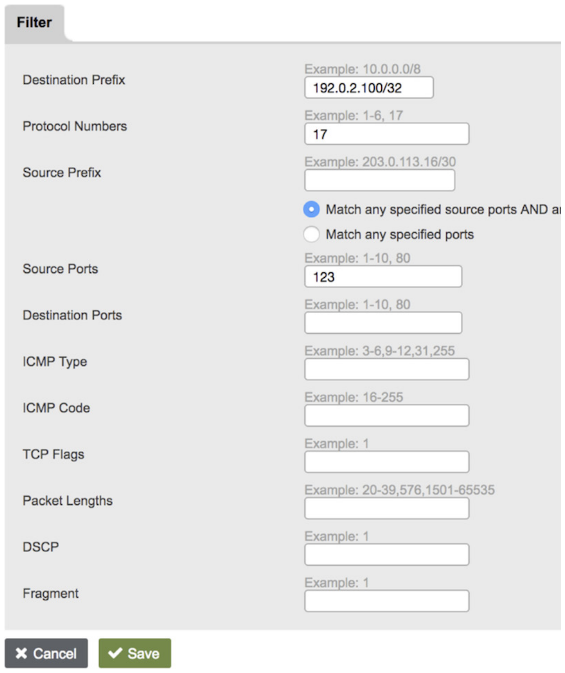
On the announcement tab use the Select Routers button to pull in the correct target and then Select Community Group to copy the correct values.



Announcement

Routers	MX-1
Community	Example: 6543:3453 129:874 — 64496:500
	[ ] Local AS
	[✓] No advertise
	[✓] No export
	[ ] No peer

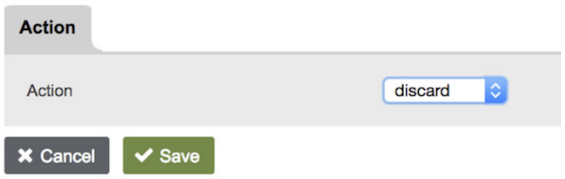
Use the GUI's Filter tab to enter your values saved from the traffic details. The following shows a typical NTP reflection attack:



Filter

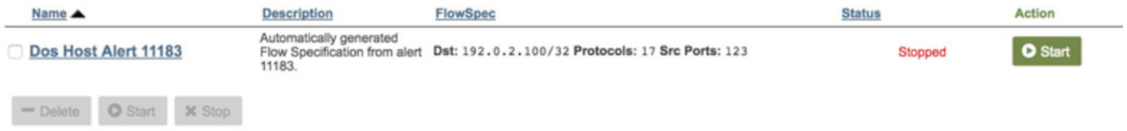
Destination Prefix	192.0.2.100/32
Protocol Numbers	17
Source Prefix	
	● Match any specified source ports AND a
	○ Match any specified ports
Source Ports	123
Destination Ports	
ICMP Type	
ICMP Code	
TCP Flags	
Packet Lengths	
DSCP	
Fragment	

Next, on the Action tab, set the discard option.


Action

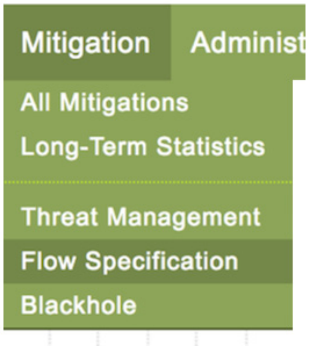
Action: discard

Now the mitigation can be controlled from the main listing screen.



Name ▲	Description	FlowSpec	Status	Action
☐ Dos Host Alert 11183	Automatically generated Flow Specification from alert 11183.	Dst: 192.0.2.100/32 Protocols: 17 Src Ports: 123	Stopped	Start

If you want to send an announcement, independent of an Alert, you can set that directly from the Mitigation menu, under Mitigation > Flow Specification.



On the MX Series, the announcement should now be visible in the inetflow routing table. Let’s check the lab MX:

```
root > show route receive-protocol bgp 192.0.2.11 table inetflow.0
inetflow.0: 1 destinations, 1 routes (1 active, 0 holddown, 0 hidden)
  Prefix      Nexthop        MED     Lclpref    AS path
  192.0.2.100,*,proto=17,srcport=123/term:1
*                          Self                          100        ?
root > show route table inetflow.0 extensive

inetflow.0: 1 destinations, 1 routes (1 active, 0 holddown, 0 hidden)
64.58.255.10,*,proto=17,srcport=53/term:1 (1 entry, 1 announced)
TSI:
KRT in dfwd;
Action(s): discard,count
```

```
                *BGP    Preference: 170/-101
                        Next hop type: Fictitious
                        Address: 0x9279f64
                        Next-hop reference count: 1
                        State: <Active Int Ext>
                        Local AS: 64496 Peer AS: 64496
                        Age: 28
                        Validation State: unverified
                        Task: BGP_64496.192.0.2.100+2271
                        Announcement bits (1): 0-Flow
                        AS path: ?
                        Communities: 64496:500 no-export no-advertise traffic-rate:0:0
                        Accepted
                        Localpref: 100
                        Router ID: 127.0.0.1
```

And we can see the effectiveness of the filter by checking the counter for the number of packets that are hitting the filter parameters:

```
root > show firewall filter __flowspec_Upstream_inet__

Filter: __flowspec_Upstream_inet__
Counters:
Name                                                Bytes              Packets
192.0.2.100,*,proto=17,srcport=123               23815402               182184
```

Recipe 10: Integrate MX Series With Arbor Networks TMS Off Ramp

Arbor Networks TMS can be added to its SP product to provide an off-ramp scrubbing station for DDoS mitigation. It requires the Arbor SP to already be successfully implemented in the network to the managed MX Series. TMS is a hardware solution with various models to accommodate the capacity of traffic required.

Problem

During a DDoS event, malicious traffic is sent to the victim along with legitimate traffic. The challenge is to drop the malicious traffic while allowing the legitimate connections to proceed as normal. Attackers use a variety of techniques to make identifying and dropping their traffic difficult.

Solution

Off-ramp scrubbing solutions provide inline processing and evaluation of the traffic to identify the attack traffic, and more surgically, to drop the malicious activity. Off ramping is used on traffic after simple firewall filters have removed the more obvious attack traffic. A wide variety of countermeasures are available and these are changed by the TMS processor, on the fly, to counteract the dynamic attack activity.

In order to evaluate and process the traffic, the off ramp appliance diverts the inbound traffic for the IP address that is under attack. Using BGP, the IP address of the host under attack is advertised to the upstream MX Series, so instead of proceeding down the normal network path, the traffic enters the Arbor TMS. After processing, the traffic is returned to the downstream network path.

In this scenario, MX-1 is the device under licensed management by the Arbor SP appliance. MX-2 represents the further downstream path for the client's Internet traffic. These can also be configured as virtual routing instances on the same MX Series platform to accomplish the necessary diversion traffic flow. The basic requirement is that there be a separation of routing tables between the upstream forwarding to the Internet peers, where the off-ramp interface is connected, and the downstream path to the client networks where the off-ramp interface will return the clean traffic to the flow, as illustrated in Figure 10.1. The IP address assignments for this recipe are listed in Table 10.1.

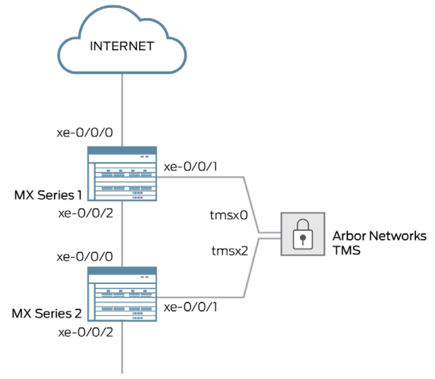


Figure 10.1 The Off-ramp Interface Returns the Clean Traffic to the Flow

Table 10.1 Table IP Address Assignments

Device	Interface	Address	Description
Arbor SP	eth0	192.0.2.11/24	BGP; SNMP
	eth1	172.16.0.11/24	jflow
MX-1 Upstream	lo0.0	127.0.0.1/32	BGP; SNMP
	fxp0.0	172.16.0.10/24	Jflow
	xe-0/0/0	198.51.100.0/31	Internet Peer
	xe-0/0/1	198.51.100.2/31	TMS On Ramp tmsx0
	xe-0/0/2	198.51.100.4/31	MX-2 Downstream
MX-2 Downstream	xe-0/0/0	198.100.100.5/31	MX-1 Upstream
	xe-0/0/1	198.51.100.6/31	TMS Off Ramp tmsx2
	xe-0/0/2	198.51.100.8/31	Downstream clients
Arbor TMS	tmsx0	198.51.100.3/31	On Ramp
	tmsx2	198.51.100.7/31	Off Ramp
	mgt0	192.0.2.12/24	BGP - mgmt

MX Series Configuration

The MX requires that the interfaces connecting to the TMS be set up in the link subnets and that a BGP peer session be established to receive the mitigation routes for the traffic that needs to be sent to the off ramp. The interface configuration is standard:

MX-1

```
[edit]
root@# set interfaces xe-0/0/1 description “Arbor TMS | tmsx0”
root@# set interfaces xe-0/0/1 unit 0 description “Upstream MX to Arbor TMS offramp”
root@# set interfaces xe-0/0/1 unit 0 family inet address 198.51.100.2/31
---These sampling commands are required on every interface of an Arbor SP managed router.
They can be applied directly or via apply-groups to all interfaces
root@# set interfaces xe-0/0/1 unit 0 family inet sampling input
root@# set interfaces xe-0/0/1 unit 0 family inet sampling output
```

MX-2

```
[edit]
root@# set interfaces xe-0/0/1 description “Arbor TMS | tmsx2”
root@# set interfaces xe-0/0/1 unit 0 description “Downstream MX from Arbor TMS onramp”
root@# set interfaces xe-0/0/1 unit 0 family inet address 198.51.100.6/31
```

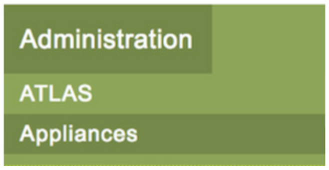
This configuration has an iBGP to the MX loopback address from the Arbor TMS mgt0 interface. These peerings are set up as standard iBGP internal peers using the local AS with the default loopback addresses. The idea is to integrate the Arbor appliances into the upstream peerings:

```
[edit]
root@# set protocols bgp group ArborTMS type internal
root@# set protocols bgp group ArborTMS family inet unicast
root@# set protocols bgp group ArborTMS peer-as 64496
root@# set protocols bgp group ArborTMS neighbor 192.0.2.12 local-address 127.0.0.1
```

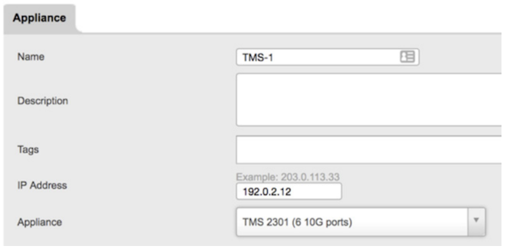
NOTE If your MX Series is using a “protect-re” firewall filter on the loopback interface, you will need to add the Arbor SP addresses to the sections that restrict SNMP and BGP traffic.

Configure TMS Appliance in Arbor SP

You will now need to add the MX Series as a managed router on the Arbor SP installation GUI. Administration > Appliances > Add Appliance button:



And add the management IP address along with the hardware type:



Appliance

Name	TMS-1
Description	
Tags	
IP Address	Example: 203.0.113.33 192.0.2.12
Appliance	TMS 2301 (6 10G ports)

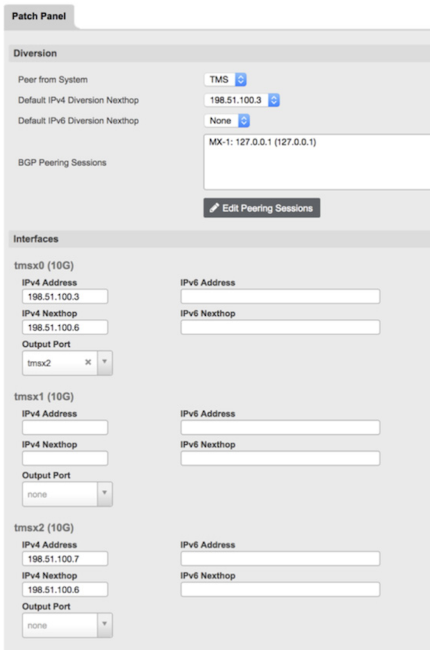
The IPv4 diversion next hop is the address that will be put into the BGP advertisement to use the off-ramp interface of the TMS. Use the pull-down Name list to select the TMS interface that acts as the on ramp.

The Patch Panel (see next page screen capture) tab settings use the interface addresses assigned to the TMS connecting to the upstream and downstream MX devices. The interface addresses come from the ones assigned to tmsx0 and tmsx2 that provide the on and off ramp connection.

The Edit Peering Sessions button allows you to select the MX device's BGP peers that are available in the Arbor system. This will be the upstream MX where you want to advertise the diversion route.

Since tmsx0 is the on ramp, set the output interface to be the off-ramp interface to the downstream router.

On both interfaces set the next hop to be the output interface on the downstream MX router. This will be the next hop that traffic leaving the TMS will use.



Patch Panel

Diversion

Peer from System	TMS
Default IPv4 Diversion Nexthop	198.51.100.3
Default IPv6 Diversion Nexthop	None
BGP Peering Sessions	MX-1: 127.0.0.1 (127.0.0.1)

Interfaces

tmsx0 (10G)

IPv4 Address	198.51.100.3
IPv4 Nexthop	198.51.100.6
Output Port	tmsx2
IPv6 Address	
IPv6 Nexthop	

tmsx1 (10G)

IPv4 Address	
IPv4 Nexthop	
Output Port	none
IPv6 Address	
IPv6 Nexthop	

tmsx2 (10G)

IPv4 Address	198.51.100.7
IPv4 Nexthop	198.51.100.6
Output Port	none
IPv6 Address	
IPv6 Nexthop	

Verifications

Let's confirm that the interfaces are up and active on the MX-1:

```
root > show interfaces descriptions | match apf
xe-0/0/1        up    up   Arbor TMS | tmsx0
xe-0/0/1.0      up    up   Upstream MX to Arbor TMS offramp
```

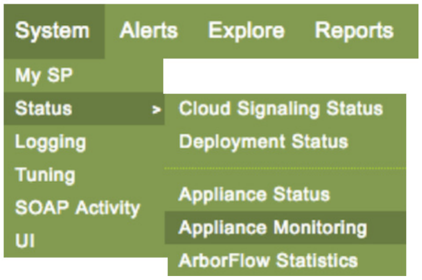
And the interfaces on MX-2:

```
xe-0/0/1        up    up   Arbor TMS | tmsx2
xe-0/0/1.0      up    up   Downstream MX from Arbor TMS onramp
```

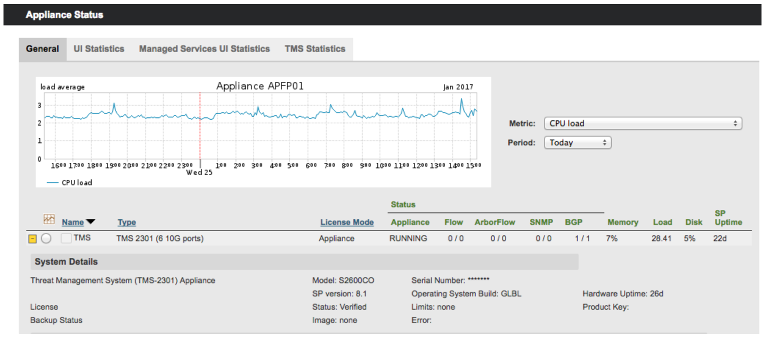
On MX-1 let's confirm the BGP peer is up and established:

```
root > show bgp neighbor 192.0.2.12
Peer: 193.0.2.11+3038 AS 33154 Local: 127.0.0.1+179 AS 64496
  Type: Internal    State: Established    Flags: <Sync RSync>
  Last State: EstabSync     Last Event: RecvKeepAlive
  Last Error: Hold Timer Expired Error
  Options: <Preference LocalAddress PeerAS Refresh>
  Local Address: 127.0.0.1 Holdtime: 90 Preference: 170
  Number of flaps: 14
  Last flap event: Closed
  Error: 'Hold Timer Expired Error' Sent: 4 Recv: 1
  Peer ID: 192.0.2.12     Local ID: 127.0.0.1      Active Holdtime: 90
  Keepalive Interval: 30         Group index: 16   Peer index: 0
  BFD: disabled, down
  NLRI for restart configured on peer: inet-unicast
  NLRI advertised by peer: inet-unicast inet-flow
  NLRI for this session: inet-unicast
  Peer does not support Refresh capability
  Stale routes from peer are kept for: 300
  Peer does not support Restarter functionality
  Peer does not support Receiver functionality
  Peer supports 4 byte AS extension (peer-as 33154)
  Peer does not support Addpath
  Table inet.0 Bit: 20007
    RIB State: BGP restart is complete
    Send state: in sync
    Active prefixes:              0
    Received prefixes:            0
    Accepted prefixes:            0
    Suppressed due to damping:    0
    Advertised prefixes:          423726
  Last traffic (seconds): Received  0    Sent 2     Checked 30
  Input messages:  Total 11289  Updates 0 Refreshes 0 Octets 214521
  Output messages: Total 906633 Updates 895002  Refreshes 0 Octets 105942039
  Output Queue[1]: 0
  Trace options: state
  Trace file: /var/log/bgp.log size 10485760 files 10
```

Let's return to the GUI and look at the status menu, by going to System > Status > Appliances.



And the TMS appliance should confirm that the BGP peer is up.



Metric: CPU load

Period: Today

			Status								
Name	Type	License Mode	Appliance	Flow	ArborFlow	SNMP	BGP	Memory	Load	Disk	SP Uptime
TMS	TMS 2301 (6 10G ports)	Appliance	RUNNING	0 / 0	0 / 0	0 / 0	1 / 1	7%	28.41	5%	22d

System Details

Threat Management System (TMS-2301) Appliance	Model: S2600CO	Serial Number: *******	
	SP version: 8.1	Operating System Build: GLBL	Hardware Uptime: 26d
License	Status: Verified	Limits: none	Product Key:
Backup Status	Image: none	Error:	

Recipe 11:
Migrate Your Core to Centralized Route Reflection and Segment Routing

The aim of this recipe is to provide sample migration scenarios of a typical service provider network moving toward two new networking enhancements in order to facilitate the integration of SD-WAN solutions.

Problem

You just got orders from the boss. Go seek out SD-WAN solutions and report back to the execs. Yikes! Well, you want to look at two things: route reflection and segment routing.

The first step is to centralize the route reflection function in your network. A route reflector (RR) solution is one widely deployed by providers to distribute BGP routes in a scaled environment. Most of time, the BGP route reflection function consists of several dedicated physical routers connected at several right places of your network. For scaling purposes the topology of the RRs is usually hierarchical: regional vs. national RRs. With the emergence of virtualization technologies, more and more often the RR's function becomes just a piece of software that can easily run on appliances. Virtual route reflecting scales out very well and the centralization of the RR's function becomes more accessible. Nevertheless, for centralizing route reflection function in a network you need to play with some tricks to help avoid routes blackholing or non-optimal routing, especially with some route families.

The second step is to switch the MPLS transport solution from LDP to Segment Routing (SPRING): SPRING seems to be one of the key drivers to deploy SDN in service providers and enterprise backbones.

This solution provides flexibility and agility to define and build LSP with or without traffic engineering (TE) constraints, and in some topologies and failure scenarios offers better fast reroute coverage than other solutions such as LFA or rLFA.

Solution 1: Centralize Route Reflection

Figure 11.1 shows you a classic service provider network with regional point-of-presence (PoPs) connected to an MPLS backbone. Each location has its regional Internet exit point made of two ASBRs connected to the Internet transit networks. The IGP is IS-IS and a unique Level 2 area composed this network. Each MX router runs Junos 16.1R2.11.

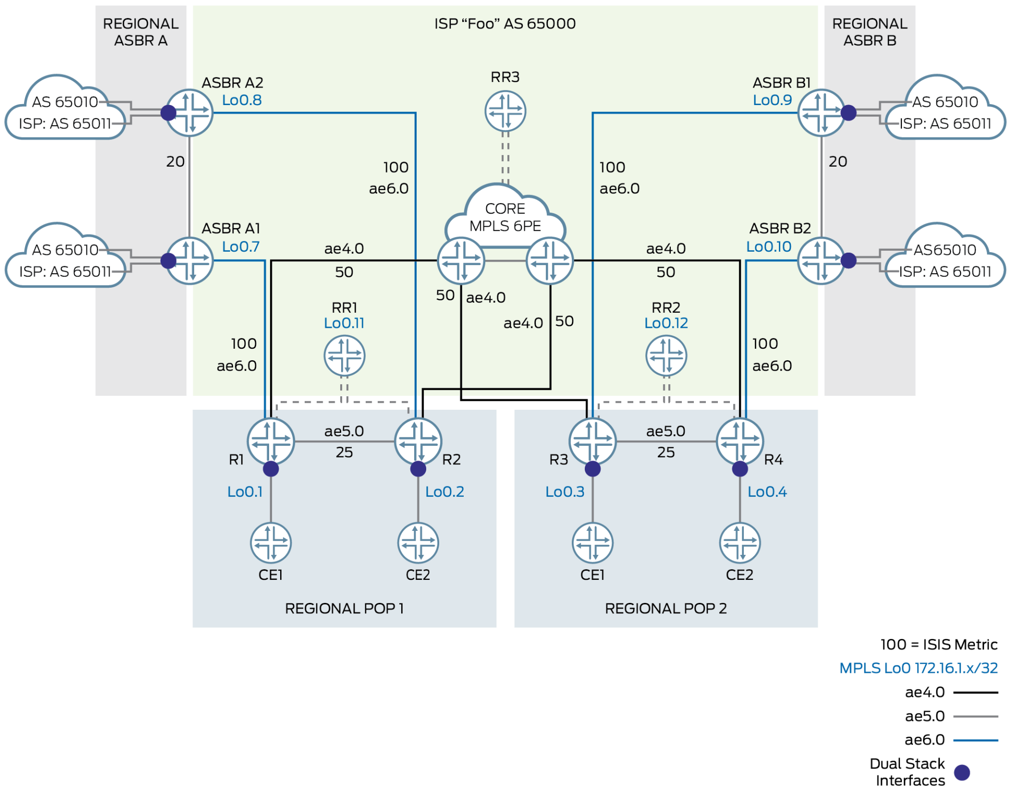


Figure 11.1: Typical Service Provider Network

To simplify the reading, Figure 11.1 represents only two regional PoPs of the “Foo” service provider. As observed, each PoP, is made of two PEs, is connected to the core network via P routers and to two ASBRs. The entire core network is a pure IS-IS/ LDP network (BGP free core). It conveys L2/L3VPN services and also offers IPv6 connectivity via the 6PE technology. A regional PoP has its dedicated RR (respectively RR1 for PoP 1, and RR2 for PoP 2). In real networks, any RR is often redundant (it means operator installs a couple of RRs). All edge routers (both ASBRs and PEs) of a given location peer with the local RR. Each local RR then peers with a national RR (RR3) in order to exchange customer’s routes of each location with the entire network. The regional default routes must not be reflected by Regional RR to National RR in order to avoid traffic attraction from one regional area over another. Only CEs can communicate with each other independent of where they are located.

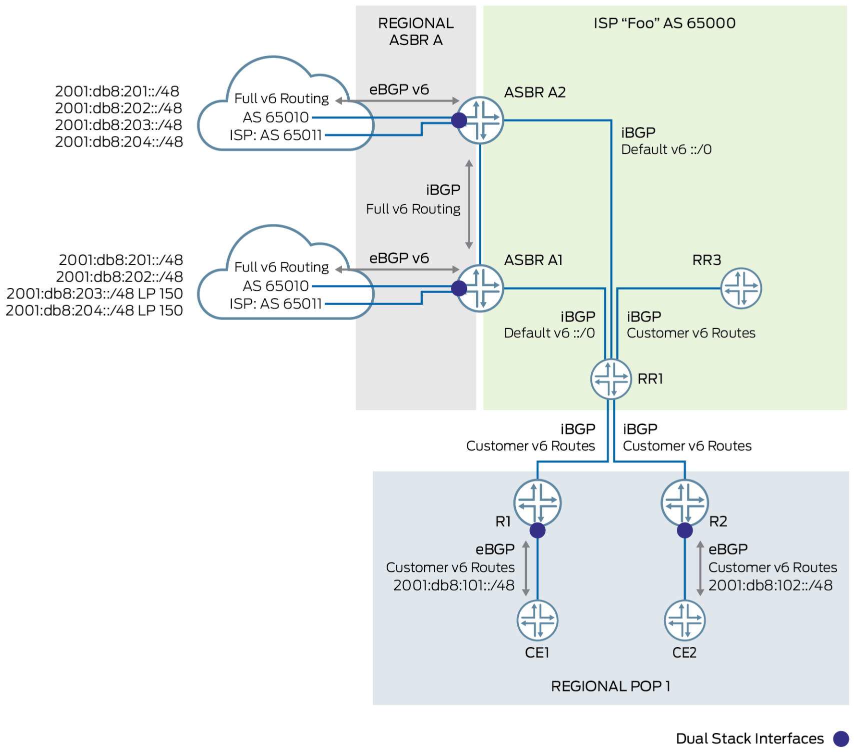


Figure 11.2 BGP Peering on PoP 1

The local "customers" of Foo are connected to the local PEs and can reach either the Internet (via ASBR) or remote customers (inter-regional traffic through the core network). Foo supports both IPv4 and IPv6 protocols. For purposes of this recipe's brevity, let's only consider the IPv6 transit service offered to customers.

The "full" IPv6 Internet routing table must reside only on the ASBRs. Each regional ASBR sends a default route to its regional PEs via the local RR. Let's zoom in to the PoP 1 (the configuration is exactly the same on PoP 2).

Figure 11.2 illustrates that all four PEs (A1, A2, R1, and R2) peer with RR1. RR1 then peers with RR3.

Let's start with RR1's configuration:

```
user@RR1> show configuration protocols bgp
log-updown;
group REGIONAL-1 {
   type internal;
   local-address 172.16.1.11;
   family inet6 {
      labeled-unicast {
         explicit-null;
      }
   }
   cluster 0.0.0.1;     <<< Unique Cluster ID per regional area
   neighbor 172.16.1.1;
   neighbor 172.16.1.2;
   neighbor 172.16.1.7;
   neighbor 172.16.1.8;
}
group NATIONAL {
   type internal;
   local-address 172.16.1.11;
   family inet6 {
      labeled-unicast {
         explicit-null;
      }
   }
   export DENY-DEFAULT;  <<< Avoid Regional default route leaking
   neighbor 172.16.1.13;
}
```

RRs are only connected to the network with the IS-IS protocol. To allow 6PE next-hop resolution, copy the inet.0 routes in the inet6.3 table like this:

```
user@RR1> show configuration routing-options
rib-groups {
   rib_isis {
      import-rib [ inet.0 inet6.3 ];
   }
}
```

Now, let's move on with the Internet access. The Internet transit networks, AS65010 and AS65011, send the full Internet routing table to ASBR A1 and A2. There is an export policy on A1 and A2 to send a default route only to RR1 and thus to regional PEs. You keep the full Internet table only on A1 and A2. There is also a dedicated iBGP session between A1 and A2 to exchange the full routing table one more time for securing the architecture. And, as observed, some routes coming from AS65011 are privileged by ASBR A1 (Local Preference 150). Here is the BGP configuration of A1:

```
user@A1> show configuration protocols bgp
log-updown;
group RR-6PE {
   type internal;
   local-address 172.16.1.7;
   family inet6 {
      labeled-unicast {
         explicit-null;
      }
   }
   export DEFAULT-V6;    <<<<<<<<<<<<< ONLY SEND the ::/0 route to RR1
   neighbor 172.16.1.13;
}
group eBGP-AS65010 {
   type external;
   peer-as 65010;
   neighbor 2001:db8:cafe::13;
}
group eBGP-AS65011 {
   type external;
   import ADD_LP150;    <<<<<<<<<<< Some routes are best by A1
   peer-as 65011;
   neighbor 2001:db8:cafe::15;
}
group INTER-ASBR {
   type internal;
   local-address 172.16.1.7;
   advertise-external;  <<<<< To avoid hidden eBGP routes due to LP 150
   family inet6 {
      labeled-unicast {
         explicit-null;
      }
   }
   neighbor 172.16.1.8;
}
```

Finally, let's have a look at the failure protection features implemented in this network. As already presented above, each ASBR has a dedicated iBGP session with its neighbor. The `advertise-external` knob is configured to force an ASBR to still send a best eBGP route to its iBGP peers even if this route is preferred by an internal path. This is the case for A2, which receives some routes from A1 with a local preference of 150. These routes are better than direct eBGP routes of A2 with a default local preference of 100. Without the knob, these eBGP routes would not be advertised by A2 to A1 and in case of a loss of the eBGP sessions on A1, the missing routes should be learned back from A2, and that could impact the convergence time.

Just to see an example, the `advertise-external` knob has been removed on A2 and now let's check the BGP paths of some of the prefixes on A1:

```
user@A1> show route 2001:db8:202::/47

inet6.0: 17 destinations, 19 routes (17 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

2001:db8:202::/48  *[BGP/170] 22:57:22, localpref 100
                      AS path: 65010 I, validation-state: unverified
                    > to 2001:db8:cafe::13 via ae10.0  <<< To Other ASs
                    [BGP/170] 00:04:40, localpref 100, from 172.16.1.8
                      AS path: 65010 I, validation-state: unverified
                    > to 192.168.1.49 via ae11.0, Push 2  <<<< To A2
2001:db8:203::/48  *[BGP/170] 01:13:22, localpref 150
                      AS path: 65011 I, validation-state: unverified
                    > to 2001:db8:cafe::15 via ae10.0 <<< To Other ASs
```

The highlighted prefix 2001:db8:203::/48 is one prefix preferred (by default) through A1 because it has a specific local preference set to 150. This route, sent by A1 to A2 via the dedicated BGP session, would be best on A2 (instead of the direct eBGP route with the default LP set to 100). In this case, the eBGP route of A2 will not be sent to A1. This is why on A1 you only see the direct eBGP route for 2001:db8:203::/48.

To force A2 to send the "backup and best" eBGP path, use the `advertise-external` feature. Once configured on A2, you can check the routes on A1 and confirm that A1 now knows both paths – one from the direct eBGP session and the other via the iBGP session with A2:

```
user@A1> show route 2001:db8:202::/47

inet6.0: 17 destinations, 21 routes (17 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

2001:db8:202::/48  *[BGP/170] 1d 01:34:25, localpref 100
                      AS path: 65010 I, validation-state: unverified
                    > to 2001:db8:cafe::13 via ae10.0
                    [BGP/170] 02:41:43, localpref 100, from 172.16.1.8
                      AS path: 65010 I, validation-state: unverified
                    > to 192.168.1.49 via ae11.0, Push 2
2001:db8:203::/48  *[BGP/170] 03:50:25, localpref 150
                      AS path: 65011 I, validation-state: unverified
                    > to 2001:db8:cafe::15 via ae10.0
                    [BGP/170] 00:00:03, localpref 100, from 172.16.1.8
                      AS path: 65011 I, validation-state: unverified
                    > to 192.168.1.49 via ae11.0, Push 2
```

On each regional RR add the `add-path send 2` knob and on each PE configure the `Add-Path receive` statement. This allows you to send more than one, but at most two, paths for each prefix the route reflector knows. At least two paths of the default routes (one per ASBR) are needed on remote PE in order to use the PIC Edge feature (see below). The RR1 and R1 `add-path` configuration are depicted here:

```
user@RR1> show configuration protocol bgp
[..]
   family inet6 {
       labeled-unicast {
           add-path {
               send {
                   path-count 2;
               }
           }
           explicit-null;
       }
   }
[..]
user@R1> show configuration protocol bgp
[..]
    family inet6 {
        labeled-unicast {
            add-path {
                receive;
            }
            explicit-null;
        }
    }
[..]
```

In order to minimize the convergence time (in case of some failure scenarios) the PE routers also have two knobs configured. First is the IPv6 `protect core` feature (aka PIC Edge). This feature installs both nominal and backup BGP paths in FIB. It protects the case of a remote next-hop failure:

```
user@R1> show configuration routing-options
rib inet6.0 {
    protect core;
}
```

The second knob feature is the LFA `link-protection`: this feature allows installation of both nominal and backup forwarding next hop in the FIB for fast reroute in case of a nominal link failure. We apply it on each core interface:

```
user@R1> show configuration protocols isis
interface ae6.0 {
    point-to-point;
    link-protection;
    level 2 metric 25;
}
```

After setting up all of these protection features, you should check on how the default route is known on PE1. And, as you can see next, the default route has two paths: one from A1, and the other from A2. Both BGP paths are installed in the FIB (PIC Edge) and for each path there are two forwarding next hops installed (LFA): the nominal interface is the R1/A1's (ae6.0) link and a backup interface computed by LFA is the R1/R2's link (ae5.0):

```
user@R1> show route logical-system R1 ::/0 exact

inet6.0: 11 destinations, 13 routes (11 active, 0 holddown, 0 hidden)
@ = Routing Use Only, # = Forwarding Use Only
+ = Active Route, - = Last Active, * = Both

::/0    @[BGP/170] 00:00:23, localpref 100, from 172.16.1.11
          AS path: I
        validation-state: unverified, > to 192.168.1.54 via ae6.0, Push 2
          to 192.168.1.2 via ae5.0, Push 2, Push 300192(top)
        [BGP/170] 00:00:23, localpref 100, from 172.16.1.11
          AS path: I
        validation-state: unverified, > to 192.168.1.54 via ae6.0, Push 2, Push 299792(top)
          to 192.168.1.2 via ae5.0, Push 2, Push 300080(top)
       #[Multipath/255] 00:00:23, metric2 1
        > to 192.168.1.54 via ae6.0, Push 2                << Nominal BGP path
          to 192.168.1.2 via ae5.0, Push 2, Push 300192(top)
          to 192.168.1.54 via ae6.0, Push 2, Push 299792(top) << Backup BGP path
          to 192.168.1.2 via ae5.0, Push 2, Push 300080(top)
```

Now it's time to migrate this network toward a centralized route reflector (RR3) for our 6PE family.

Migrate To a Centralized RR

An RR reflects a route to its BGP clients by default only if that route is the best from the RR's point of view. In other words, if the RR receives several BGP paths of a given prefix, it picks the best path and only reflects that path to its clients. BGP tiebreakers are used to select the best path, and most of the time the selector is the Interior Gateway Protocol (IGP) distance to the next hop. Because the cost to a given point in the network will vary across routers, and, conversely, RRs are usually outside of the forwarding plane – but still connected to the network – the IGP metric as best path selector is often not relevant. As a consequence, the choice of exit point for a RR and its clients will be the best exit point for the RR - not necessarily the one best for the RR clients. It is important to note that this drawback does not impact all route families the same way. For instance, L3VPN routes are usually unique in terms of "path". That's because an L3VPN route is the concatenation of the prefix itself with a `route-distinguisher` that is often uniquely attributed by the operator per PE or per VRF. In this way, for the RR, all L3VPN prefixes are unique (have a unique path) and thus are distributed to the clients.

NOTE To optimize the L3VPN route distribution use the Route Target Constraint solution, which is unfortunately beyond the scope of this recipe.

As shown in Figure 11.1, to avoid the current limitations, the service provider Foo had chosen the placement of physical route reflector equipment on each regional area. This meant installing several RRs in several IGP locations of the network to indirectly force the RR to select the closest exit point of a given network region. In this configuration, the RRs should be meshed to each other to allow the distribution of regional routes in the entire network. A second layer of route reflection is often used to aggregate and reflect routes from regional RRs – in this case study, RR1 and RR2 peer with an upper stage of RR: RR3.

Evaluate Solutions for Centralizing RR

To centralize the RR function, the first solution consists of sending all paths or a subset of paths of each prefix to all RR's clients. In this case, the best path selection would be accomplished by the client itself. This approach – which uses what is called *BGP Add-Path feature* – suffers the significant drawback of pushing a large amount of BGP states to all edge routers. Moreover, `add-path` is also limited in terms of the number of paths it can send. For instance, the Junos OS currently supports six paths. In our configuration, where there are several regional network areas (more than two), six paths wouldn't be enough. However, add-path has, in other circumstances, had the opportunity to offer path diversity to BGP clients (i.e. for better convergence time in a failure scenario – see above in addition with PIC Edge).

This recipe details a new way to distribute routes via a RR independent of where is the RR located. The core of this solution is to configure the RR to use a virtual IGP location for a given BGP client or a group of clients (aka peer group). This enables having a given group of clients receive routes with optimal distance to the next hops from the position of the configured virtual IGP location. This solution is called: *Optimal Route Reflection* (ORR), and has been implemented in Junos since 15.1.

This solution will be deployed in the national route reflector RR3. Let's start to modify our network configuration.

Migration Procedure

As each network has its "change" processes, let's not cover the step-by-step migration from several regional RRs to one national RR. You might imagine several scenarios, such as double meshing PEs and ASBRs with the regional RR and the rational RR, then shutting down the BGP session to the regional RR, and so on.

Instead, let's focus on what happens if the RR function is centralized without doing anything more than establishing BGP sessions between our remote BGP RR clients and RR3, and then how ORR can help solve the issue.

ORR Theory

In theory, ORR is quite simple, beginning with virtual IGP location. That means the capability of a router to run an SPF not from its point of view, its own placement in the IGP topology, but by choosing another remote node as if it was the root of the SPF tree. The Junos ORR configuration is very simple. It only takes in the input of the router ID of the remote node – the one you want to compute the SPF. You could also provide a backup router ID in case the primary remote node fails and disappears off the IGP database. Figure 11.3 illustrates these ORR concepts.

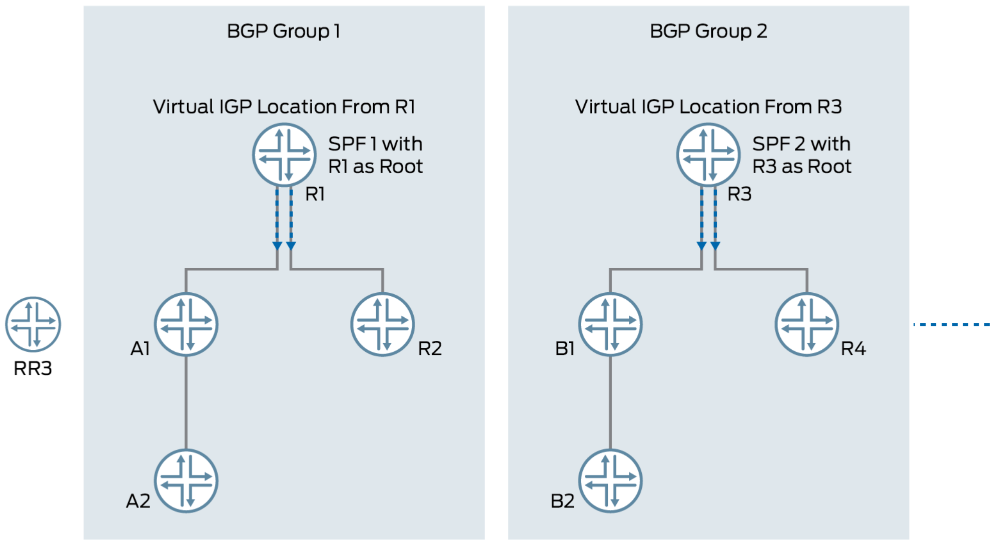


Figure 11.3 ORR Concepts

Changing Our Topology

Let's assume that all BGP sessions between A1, A2, R1, R2, B1, B2, R3, and R4 are established with RR3, but RR1 and RR2 have been removed. Figure 11.4 provides some information regarding the routes sent by the RR's clients to RR3 – the four ASBRs send only the default IPv6 route and each PE sends its customer an attached prefix.

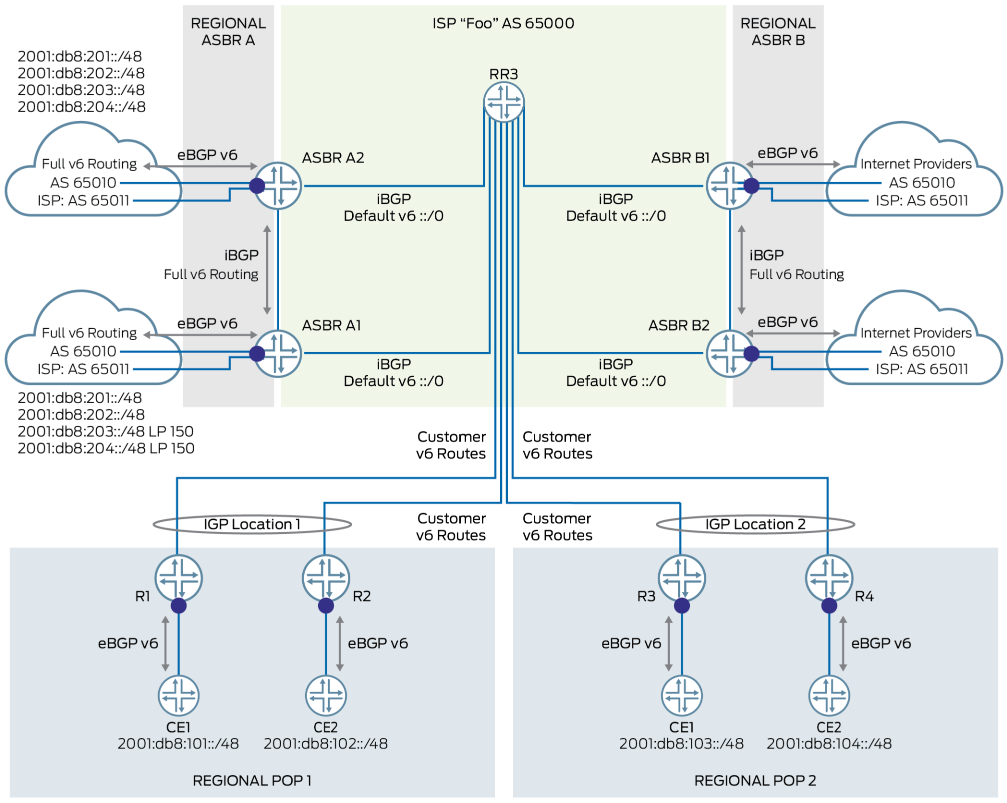


Figure 11.4 Move BGP Sessions to National RR

For practical purposes, let's not configure ORR in the BGP groups yet, so we can observe the issue you might encounter with a centralized RR. Given that bit of news, the RR3 router is configured like this:

```
user@RR3> show configuration protocols bgp
log-updown;
group REGIONAL-1 {
    type internal;
    local-address 172.16.1.13;
    family inet6 {
        labeled-unicast {
            explicit-null;
        }
    }
    cluster 0.0.0.3;
    neighbor 172.16.1.1;
    neighbor 172.16.1.2;
    neighbor 172.16.1.7;
```

```
   neighbor 172.16.1.8;
}
group REGIONAL-2 {
   type internal;
   local-address 172.16.1.13;
   family inet6 {
      labeled-unicast {
         explicit-null;
      }
   }
   cluster 0.0.0.3;
   neighbor 172.16.1.3;
   neighbor 172.16.1.4;
   neighbor 172.16.1.9;
   neighbor 172.16.1.10;
}
```

You can see a separate BGP group per regional network location. In this configuration RR3 should receive four paths of the default IPv6 route and one path for each customer's prefix. Let's check:

```
user@RR3> show route protocol bgp

::/0              *[BGP/170] 1d 23:52:43, localpref 100, from 172.16.1.7 << Best path from A1
                    AS path: I, validation-state: unverified
                  > to 192.168.1.130 via ae5.0, Push 2
                  [BGP/170] 23:00:55, localpref 100, from 172.16.1.8
                    AS path: I, validation-state: unverified
                  > to 192.168.1.134 via ae8.0, Push 2
                  [BGP/170] 1d 23:52:43, localpref 100, from 172.16.1.9
                    AS path: I, validation-state: unverified
                  > to 192.168.1.130 via ae5.0, Push 2
                  [BGP/170] 1d 23:46:18, localpref 100, from 172.16.1.10
                    AS path: I, validation-state: unverified
                  > to 192.168.1.134 via ae8.0, Push 2
2001:db8:101::/48  *[BGP/170] 1d 23:52:43, localpref 100, from 172.16.1.1
                    AS path: 65100 I, validation-state: unverified
                  > to 192.168.1.130 via ae5.0, Push 2
2001:db8:102::/48  *[BGP/170] 1d 23:46:18, localpref 100, from 172.16.1.2
                    AS path: 65100 I, validation-state: unverified
                  > to 192.168.1.134 via ae8.0, Push 2
2001:db8:103::/48  *[BGP/170] 1d 23:52:43, localpref 100, from 172.16.1.3
                    AS path: 65100 I, validation-state: unverified
                  > to 192.168.1.130 via ae5.0, Push 2
2001:db8:104::/48  *[BGP/170] 1d 23:46:18, localpref 100, from 172.16.1.4
                    AS path: 65100 I, validation-state: unverified
                  > to 192.168.1.134 via ae8.0, Push 2
```

Here you can see the issue with a national RR – RR3 selects one path among the four default routes based on its IGP location viewpoint. In this case, it selects the default route sent by A1 and this route is reflected to all remote clients. For R1 and R2, this isn't a big issue. They will receive the default route coming from a next hop of their network location. But for R3 and R4 it's another story. For them, the next hop points to an exit not in their regional area. To reach the Internet they should use core backbone links. This causes the use of un-provisioned bandwidth and could trigger traffic congestion within the core. You can confirm this issue by checking which default route is sent to R1, R2, R3, and R4 from RR3. The next-hop address for the default route is 172.16.1.7 (A1) for all of them:

```
user@RR3> show route advertising-protocol bgp 172.16.1.1 ::/0 exact

inet6.0: 5 destinations, 8 routes (5 active, 0 holddown, 0 hidden)
  Prefix                 Nexthop            MED    Lclpref    AS path
* ::/0                   ::ffff:172.16.1.7          100       I

user@RR3> show route advertising-protocol bgp 172.16.1.2 ::/0 exact

inet6.0: 5 destinations, 8 routes (5 active, 0 holddown, 0 hidden)
  Prefix                 Nexthop            MED    Lclpref    AS path
* ::/0                   ::ffff:172.16.1.7          100       I

user@RR3> show route advertising-protocol bgp 172.16.1.3 ::/0  exact

inet6.0: 5 destinations, 8 routes (5 active, 0 holddown, 0 hidden)
  Prefix                 Nexthop            MED    Lclpref    AS path
* ::/0                   ::ffff:172.16.1.7          100       I

user@RR3> show route advertising-protocol bgp 172.16.1.4 ::/0 exact

inet6.0: 5 destinations, 8 routes (5 active, 0 holddown, 0 hidden)
  Prefix                 Nexthop            MED    Lclpref    AS path
* ::/0                   ::ffff:172.16.1.7          100       I
```

It's time to solve this issue by adding the ORR configuration on RR3 for each peer group, like this:

```
user@RR3> show configuration protocols bgp
log-updown;
group REGIONAL-1 {
[...]
    optimal-route-reflection {
        igp-primary 172.16.1.1;
        igp-backup 172.16.1.2;
    }
[...]
}
group REGIONAL-2 {
[...]
    optimal-route-reflection {
        igp-primary 172.16.1.3;
        igp-backup 172.16.1.4;
    }
[...]
}
```

As you can see, for each BGP group you add the specific ORR configuration. For the `REGIONAL-1` group you configure the primary virtual IGP location with the IP address of R1 (its router ID) and in case this one fails, there's a secondary virtual IGP location with the IP address of R2. You do exactly the same for the second BGP group with R3 as the primary virtual IGP location and R4 as a backup. You can then check the "virtual SPF" result for a given group:

```
user@RR3> show isis bgp-orr group REGIONAL-1
BGP ORR Peer Group: REGIONAL-1
  Primary: 172.16.1.1, active
  Backup: 172.16.1.2
IPv4/IPv6 ORR Routes
--------------------
Prefix              L Version   Metric Type
172.16.1.1/32         2     13       0 int
172.16.1.2/32         2     13      25 int
172.16.1.3/32         2     13     100 int
172.16.1.4/32         2     13     115 int
172.16.1.5/32         2     13      50 int
172.16.1.6/32         2     13      65 int
172.16.1.7/32         2     13     100 int  <<< Cost from R1 to reach A1
172.16.1.8/32         2     13     124 int
172.16.1.9/32         2     13     200 int
172.16.1.10/32        2     13     215 int
172.16.1.11/32        2     13     500 int
172.16.1.12/32        2     13     600 int
[...]

user@RR3> show route 172.16.1.7 protocol isis

inet.0: 34 destinations, 34 routes (34 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

172.16.1.7/32      *[IS-IS/18] 00:06:28, metric 650
                  > to 192.168.1.130 via lt-4/0/0.200
```

This output shows you the result of the SPF computation as if the RR3 was R1. Indeed, from RR3's point of view the loopback address for a remote ASBR such as A1 – 172.16.1.7 – is reachable with a cost of 650 while from R1 point of view it is reachable with a metric of 100. *That is the power of ORR.*

So now let's check back to see which default route is sent by RR3 to R1, R2, R3, and R4:

```
user@RR3> show route advertising-protocol bgp 172.16.1.1 ::/0 exact

inet6.0: 5 destinations, 8 routes (5 active, 0 holddown, 0 hidden)
  Prefix                Nexthop              MED     Lclpref    AS path
* ::/0                  ::ffff:172.16.1.7            100        I

user@RR3> show route advertising-protocol bgp 172.16.1.2 ::/0 exact

inet6.0: 5 destinations, 8 routes (5 active, 0 holddown, 0 hidden)
  Prefix                Nexthop              MED     Lclpref    AS path
* ::/0                  ::ffff:172.16.1.7            100        I

user@RR3> show route advertising-protocol bgp 172.16.1.3 ::/0  exact

inet6.0: 5 destinations, 8 routes (5 active, 0 holddown, 0 hidden)
  Prefix                Nexthop              MED     Lclpref    AS path
  ::/0                 ::ffff:172.16.1.10            100        I

user@RR3> show route advertising-protocol bgp 172.16.1.4 ::/0 exact
```

```
inet6.0: 5 destinations, 8 routes (5 active, 0 holddown, 0 hidden)
  Prefix                Nexthop              MED     Lclpref    AS path
  ::/0                 ::ffff:172.16.1.10           100       I
```

Interesting, isn't it? As you can see, RR3 reflects to R1 and R2, the default route of A1, and for routers R3 and R4, the default route of B2. Each regional PE receives an Internet exit point attached to its location. To keep consistent with the previous configuration with regional RR, you could add `add-path` in conjunction with ORR – then you could reflect both ASBR's default routes of each location. Let's configure this feature on RR3 (don't forget to enable add-path on remote PEs):

```
user@RR3> show configuration protocols bgp
log-updown;
group REGIONAL-1 {
    type internal;
    local-address 172.16.1.13;
    family inet6 {
        labeled-unicast {
            add-path {
                send {
                    path-count 2;
                }
            }
            explicit-null;
        }
    }
    cluster 0.0.0.3;
    optimal-route-reflection {
        igp-primary 172.16.1.1;
        igp-backup 172.16.1.2;
    }
[...]
```

Finally, let's check which default routes are sent to remote PEs one more time:

```
user@RR3> show route advertising-protocol bgp 172.16.1.1 ::/0 exact

inet6.0: 5 destinations, 8 routes (5 active, 0 holddown, 0 hidden)
  Prefix                Nexthop              MED     Lclpref    AS path
* ::/0                  ::ffff:172.16.1.7            100       I
                       ::ffff:172.16.1.8            100       I

user@RR3> show route advertising-protocol bgp 172.16.1.2 ::/0 exact

inet6.0: 5 destinations, 8 routes (5 active, 0 holddown, 0 hidden)
  Prefix                Nexthop              MED     Lclpref    AS path
* ::/0                  ::ffff:172.16.1.7            100       I
                       ::ffff:172.16.1.8            100       I

user@RR3> show route advertising-protocol bgp 172.16.1.3 ::/0 exact

inet6.0: 5 destinations, 8 routes (5 active, 0 holddown, 0 hidden)
  Prefix                Nexthop              MED     Lclpref    AS path
  ::/0                 ::ffff:172.16.1.10           100       I
                       ::ffff:172.16.1.9            100       I
```

```
user@RR3> show route advertising-protocol bgp 172.16.1.4 ::/0 exact

inet6.0: 5 destinations, 8 routes (5 active, 0 holddown, 0 hidden)
  Prefix                Nexthop              MED     Lclpref    AS path
  ::/0                  ::ffff:172.16.1.10           100        I
                        ::ffff:172.16.1.9            100        I
```

Fantastic! Each remote PE now receives its two local ASBR routes. Let's validate on R1 that you find exactly the same default route with the same protection mechanisms enabled (PIC Edge and LFA):

```
user@R1> show route ::/0 exact

inet6.0: 11 destinations, 13 routes (11 active, 0 holddown, 0 hidden)
@ = Routing Use Only, # = Forwarding Use Only
+ = Active Route, - = Last Active, * = Both

::/0                @[BGP/170] 00:17:56, localpref 100, from 172.16.1.13
                       AS path: I
                    validation-state: unverified, > to 192.168.1.54 via ae6.0, Push 2
                       to 192.168.1.2 via ae5.0, Push 2, Push 299840(top)
                    [BGP/170] 00:17:56, localpref 100, from 172.16.1.13
                       AS path: I
                    validation-state: unverified, > to 192.168.1.54 via ae6.0, Push 2, Push
299856(top)
                       to 192.168.1.2 via ae5.0, Push 2, Push 299856(top)
                  #[Multipath/255] 00:17:56, metric2 1
                    > to 192.168.1.54 via ae6.0, Push 2
                       to 192.168.1.2 via ae5.0, Push 2, Push 299840(top)
                       to 192.168.1.54 via ae6.0, Push 2, Push 299856(top)
                       to 192.168.1.2 via ae5.0, Push 2, Push 299856(top)
```

ORR Current Limitations

As seen earlier in this recipe, ORR sends the best route to a regional network (a given BGP group) from a given virtual IGP location. It may not be exactly the best route for each router (RR client) of this location. To achieve that ideal state, you could instead configure ORR per neighbor, which would occupy high CPU and memory usage on the RR. Nevertheless, you could partially solve the issue by adding `add-path` in conjunction with ORR as has been demonstrated.

The current drawback of ORR implementation is that you can't manage the ORR feature itself with a policy. The IETF documentation has described this option although it is not yet implemented in the Junos OS (but it should be put in soon).

Now let's move on to the second migration and replace LDP with Segment Routing while maintaining the same protection features (PIC Edge and LFA).

Solution 2: Migrate to Segment Routing

This recipe will not present the theory of Segment Routing (SPRING) or else it would quadruple in size. Many books cover this topic in depth. The best is the awesome *MPLS in the SDN Era*, by Sánchez-Monge and Szrkowicz, 2016, O'Reilly Media: http://www.juniper.net/us/en/training/jnbooks/oreilly-juniper-library/mpls-sdn-era/.

Current Junos Limitation with SPRING

Currently, Junos 16.1 does not support the LDP mapping server, which allows some routers to play the role of an LDP/SPRING gateway. This feature is often requested when you deploy segment routing in a network with some legacy equipment that does not support SPRING, or equipment that does not yet support the feature. It should be on the roadmap soon.

Another limitation that you may be aware of is the fact that the node-segment ID is applied to the primary loopback address. Usually the operator configures several loopback addresses (one primary and several secondary) in order to allocate a loopback per BGP family: one for BGP L3VPN, and another for BGP IPv4 or BGP 6PE. If your current loopback used by LDP as a local FEC is not the primary address, the migration from LDP to Segment Routing may not be smooth. No worries, this current limitation has been handled by Juniper and you should be able to configure which loopback address you want to be the node segment ID in a coming next release (rumors say 17.2). Junos also plans to support prefix-SID in this same release. Please note that currently SPRING is only supported on IS-IS for Junos.

Finally, Junos does not yet support TI-LFA but implements segment routing for LFA and rLFA protection mechanisms, which already covers many topologies.

Migrate the Topology

Okay, with all the disclaimers completed, let's re-examine our network diagram in Figure 11.5.

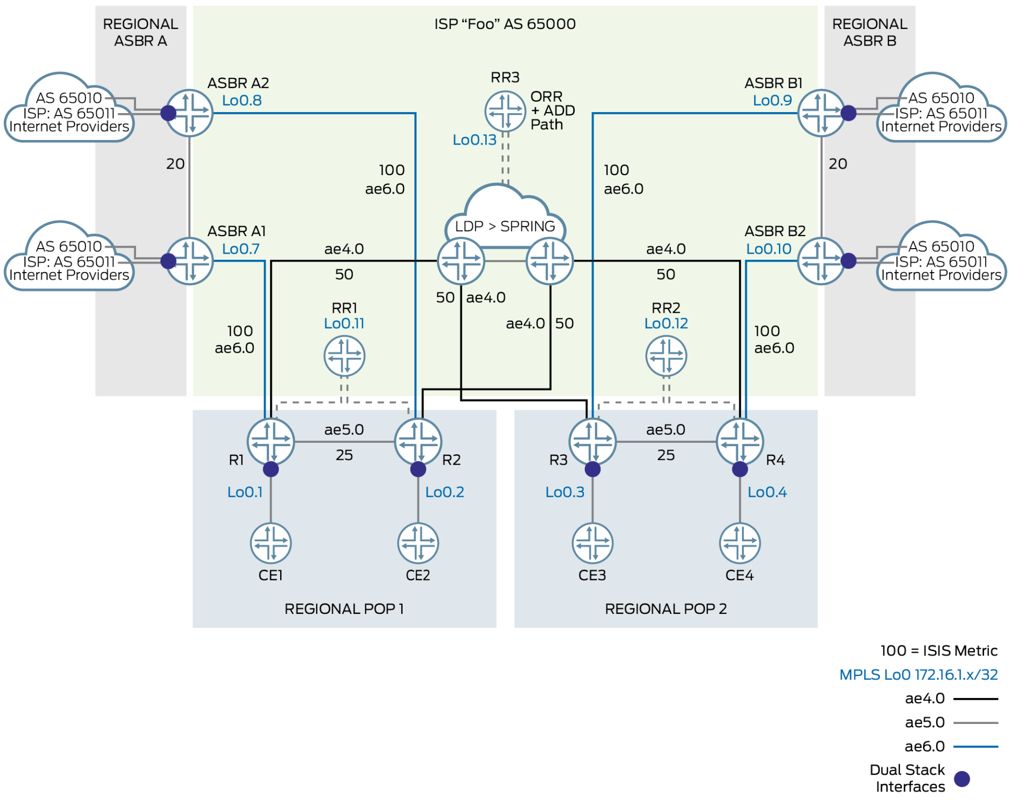


Figure 11.5 Segment Routing Migration

The first step in the migration phase is to enable segment routing on each router in the backbone. You will see it's very simple. Moreover, in this recipe's case there is only one primary loopback address that is also the local LDP FEC, so you don't need to reallocate the primary statement to the right address.

Let's allocate one unique node segment ID per router. For this case, the ID is equal to the less significant byte of the loopback address. Let's configure the `node-segment` ID on each router. Although the configuration is only shown on two nodes, R1 and R4, it's exactly the same way on the others:

```
user@R1> show configuration protocols isis
lsp-lifetime 65500;
overload timeout 60;
source-packet-routing {
    node-segment ipv4-index 1;
}
[...]
```

```
user@R4> show configuration protocols isis
lsp-lifetime 65500;
overload timeout 60;
source-packet-routing {
   node-segment ipv4-index 4;
}
[...]
```

Once segment routing is enabled everywhere let's check the inet.3 table, say on R1:

```
user@R1> show route table inet.3

inet.3: 11 destinations, 20 routes (11 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

172.16.1.2/32     *[LDP/9] 01:05:41, metric 1
                  > to 192.168.1.2 via ae5.0
                    to 192.168.1.10 via ae4.0, Push 299808
                  [L-ISIS/14] 01:05:41, metric 25
                  > to 192.168.1.2 via ae5.0
                    to 192.168.1.10 via ae4.0, Push 800002
172.16.1.3/32     *[LDP/9] 01:05:41, metric 1
                  > to 192.168.1.10 via ae4.0, Push 299888
                    to 192.168.1.2 via ae5.0, Push 299952
                  [L-ISIS/14] 01:05:41, metric 100
                  > to 192.168.1.10 via ae4.0, Push 800003
                    to 192.168.1.2 via ae5.0, Push 800003
172.16.1.4/32     *[LDP/9] 01:05:41, metric 1
                  > to 192.168.1.10 via ae4.0, Push 299936
                    to 192.168.1.2 via ae5.0, Push 299968
                  [L-ISIS/14] 01:05:35, metric 115
                  > to 192.168.1.10 via ae4.0, Push 800004
                    to 192.168.1.2 via ae5.0, Push 800004
[...]
```

As you can see, each loopback address is known by LDP and the segment routing protocol (L-ISIS). One thing to notice is the administrative distance of each protocol. By default, LDP is better than SPRING. You could confirm that by issuing a traceroute from R1 to R4:

```
user@R1> traceroute 172.16.1.4 source 172.16.1.1
traceroute to 172.16.1.4 (172.16.1.4) from 172.16.1.1, 30 hops max, 52 byte packets
 1  192.168.1.10  0.907 ms  0.830 ms  0.819 ms
     MPLS Label=299936 CoS=0 TTL=1 S=1
 2  192.168.1.26  0.717 ms  0.775 ms  0.806 ms
     MPLS Label=299888 CoS=0 TTL=1 S=1
 3  172.16.1.4  0.719 ms  1.181 ms  0.820 ms
```

The segment routing default label space starts at 800000. You can see that the LDP label is still used over the entire path. To switch to the segment routing data plane you need to configure a higher preference for LDP on each ingress PE. Please notice that switching from LDP to segment routing can be done ingress PE by ingress PE, and both data planes can work together. The only mandatory step is to enable segment routing everywhere in your network.

Let's switch the R1 PE to the segment routing data plane:

```
user@R1# set logical-systems R1 protocols ldp preference 15
          And now issue back the traceroute from R1 to R4:
user@R1> traceroute 172.16.1.4 source 172.16.1.1
traceroute to 172.16.1.4 (172.16.1.4) from 172.16.1.1, 30 hops max, 52 byte packets
 1  192.168.1.10  0.980 ms  0.799 ms  0.797 ms
    MPLS Label=800004 CoS=0 TTL=1 S=1
 2  192.168.1.26  0.730 ms  0.782 ms  0.774 ms
    MPLS Label=800004 CoS=0 TTL=1 S=1
 3  172.16.1.4  0.680 ms  0.819 ms  0.810 ms
```

Great! As ingress PE R1 now uses the transport label (in the 800K range) provided by segment routing, all the other routers also swap the segment routing label hop-by-hop until the packet reaches R4. Remember, R4 has not switched to Segment Routing so the return traffic from R4 to R1 still uses the LDP data plane.

Finally let's check back on how our default route is protected on R1:

```
user@R1> show route ::/0 exact

inet6.0: 11 destinations, 13 routes (11 active, 0 holddown, 0 hidden)
@ = Routing Use Only, # = Forwarding Use Only
+ = Active Route, - = Last Active, * = Both

::/0            @[BGP/170] 00:13:35, localpref 100, from 172.16.1.13
                  AS path: I
                validation-state: unverified, > to 192.168.1.54 via ae6.0, Push 2
                  to 192.168.1.2 via ae5.0, Push 2, Push 800007(top)
                [BGP/170] 00:13:35, localpref 100, from 172.16.1.13
                  AS path: I
                validation-state: unverified, > to 192.168.1.54 via ae6.0, Push 2, Push
800008(top)
                  to 192.168.1.2 via ae5.0, Push 2, Push 800008(top)
               #[Multipath/255] 00:13:35, metric2 100
                > to 192.168.1.54 via ae6.0, Push 2
                  to 192.168.1.2 via ae5.0, Push 2, Push 800007(top)
                  to 192.168.1.54 via ae6.0, Push 2, Push 800008(top)
                  to 192.168.1.2 via ae5.0, Push 2, Push 800008(top)
```

As you can see PIC Edge and LFA work perfectly with segment routing.

Discussion

This recipe has briefly covered two new networking enhancements that can help you to migrate progressively to SD-WAN solutions: centralizing the routing information delivering with ORR, and segment routing to easily manage the end-to-end flow forwarding. These solutions, which are quite new, are already running well on the Junos OS. Each feature will be improved upon in order to handle more complex topologies and to provide more protection mechanism.