Engineering Simplicity

Day One Million

DAY ONE: JUNIPER AMBASSADORS' COOKBOOK FOR 2019

How-to solutions and network recipes
by the community dedicated to
up and running Juniper reliability.

An Ambassador Meetup at NXTWORK 2018

By Nupur Kanoi, Chris Parker, Christian Scholz, Dan Hearty, Michel Tepper, Said van de Klundert, Paul Clarke, Martin Brown, Peter Klimai, Tom Dwyer, Pierre-Yves Maunier, Yasmin Lara, Stefan Fouant, Steve Puluka, Nick Ryce

DAY ONE: JUNIPER AMBASSADORS' COOKBOOK FOR 2019

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, LinkedIn, and other social networks.

"This is another great addition to the bulging Juniper Day One library. The Juniper Ambassadors have created recipes for a considerable number of hot-button issues facing engineers and architects today. The recipes explain and guide you through the setup of the solutions and help simplify and speed up the time to operations. There's a great mix of traditional CLI-based implementation as well the new world of automation and network reliability engineering. I'll certainly be using this book for some of my forthcoming data center and multicloud projects."
Bhupen Mistry, Director & Principal Consultant, Operativity Ltd.

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

- Virtualizing Routers with Routing Instances
- Saving Time with Apply-stuff
- Enabling the inet.3 Table for BGP to Use Label-Switched Paths
- Forcing Non-BGP Traffic to Take an LSP: Manipulating inet.3
- Setup, Best Practices, and Pitfalls of MC-LAG on the QFX-Series
- Connecting an SRX Cluster to a VRRP Router
- Consolidation of Two PEs: BGP Pre-and Post-Check With PyEz
- L2VPN to VPLS Stitching
- Configuring EVPN VLAN-Aware Bundle Service on Juniper MX
- Configuring EVPN VLAN Bundle Service on Juniper MX
- Using Terminating Actions in Junos Routing Policy
- ZTP with SLAX on EX Series Devices
- Configuring NAT on SRX Platforms Using Proxy ARP/ND
- Q-in-Q Tunneling Using ELS
- Low-Risk Methodology for Deploying Firewall Filters
- Translating RSVP-signaled LSPs for Quick Troubleshooting Using PyEZ
- Writing eBGP Policies for Outbound Traffic Engineering
- Synchronizing Junos Device Configurations Using Python Scripts
- Migrating from MC-LAG to ESI-LAG

ISBN 978-1941441916
52500
9 781941 441916

JUNIPer NETWORKS

Day One: Ambassadors' Cookbook for 2019

by Nupur Kanoi, Chris Parker, Christian Scholz, Dan Hearty, Michel Tepper, Said van de Klundert, Paul Clarke, Martin Brown, Peter Klimai, Tom Dwyer, Pierre-Yves Maunier, Yasmin Lara, Stefan Fouant, Steve Puluka, Nick Ryce

Published by Juniper Networks Books
Authors: Nupur Kanoi, Chris Parker, Christian Scholz, Dan Hearty, Michel Tepper, Said van de Klundert, Paul Clarke, Martin Brown, Peter Klimai, Tom Dwyer, Pierre-Yves Maunier, Yasmin Lara, Stefan Fouant
Technical Editors: Steve Puluka, Nick Ryce
Editor in Chief: Patrick Ames
Copyeditor and Proofer: Nancy Koerbel
Illustration: Karen Joice
Ambassador Program Manager: Julie Wider

ISBN: 978-1-941441-90-9 (print)
Printed in the USA by Vervante Corporation.
ISBN: 978-1-941441-91-6 (ebook)
Version History: v1, April 2019
 2 3 4 5 6 7 8 9 10

Visit the entire *Day One* library:
http://www.juniper.net/dayone

Comments? Error reports? Email them to dayone@juniper.net.

Contributing Ambassadors

Nupur Kanoi (Recipe 7) is a Senior Backbone engineer for a global service provider, where she has gained experience in service provider backbone architecture and design. She also holds JNCIE-ENT (#520), JNCIE-SP (#2824), JNCIP-DC, and JNCDS-DC certifications. Nupur can be reached on LinkedIn (linkedin.com/in/nupur-kanoi-520) and on Twitter (@nupur_kanoi).

Chris Parker (Recipes 3, 4, and 11) is a British network engineer, though he prefers to think of himself as a citizen of the Internet. He has worked in the service provider sector for over 10 years and has a particular passion for BGP, MPLS, and IS-IS. He holds six Juniper certifications and is currently working towards JNCIE-SP. He blogs at NetworkFunTimes.com, where he teaches Juniper technologies with a smile and a sense of humor. Find him on Twitter at @NetworkFunTimes.

Christian Scholz (Recipes 5, 12) is a German Senior Consultant for networking and security, and a Juniper Ambassador, who is currently working for a large System Integrator based in Cologne. Christian is certified JNCIE-SEC #374 and has several years of experience with migrations from any vendor to Juniper. He was involved in many large projects for German Campus Networks and eventually made his hobby his job. He enjoys blogging and giving knowledge back to the community, especially with "hot topics" like EVPN, automation, and IPv6.

Dan Hearty (Recipes 9 and 10) is a Principal Engineer working for Telent in the UK and is a Juniper Ambassador. He has over 10 years of experience specializing in service provider, data center and security technologies. He is JNCIE-SP #2406 and more recently achieved JNCIE-DC #190. Dan enjoys blogging and has a particular interest in all things EVPN.

Michel Tepper (Recipes 2 and 6) is a Solutions Architect for Nuvias in the Netherlands and a Juniper Ambassador. Besides doing a lot of pre-sales support he also is a Juniper Networks Certified Instructor on Security, Service Provider, and Enterprise Routing and Switching tracks. Working 30+ years in the industry doesn't reduce his enthusiasm for it, or for Juniper, to be specific. Besides Juniper certifications, Michel holds certifications for a number of other leading vendors.

Said van de Klundert (Recipe 16) is a Dutch networking enthusiast, Juniper Networks Ambassador, network engineer at IBM Cloud, and content developer at iNET ZERO. He has over 10 years of experience working in service provider, data center, and large cloud networks. He has a passion for network automation and holds the JNCIE-SP #2573 and JNCIE-DC #26.

Paul Clarke (Recipes 8 and 14) is a Customer Solutions Architect working for Fujitsu in the UK and is a Juniper Ambassador. He specialises in service provider, but also operates across the data center, enterprise, and security towers. He has over 20 years of experience focused on networking and holds four JNCIP certifications.

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 providers. 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.

Peter Klimai (Recipe 18) 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.

Tom Dwyer (Recipe 19) is a Principal Engineer leading the Data Center Practice at Nexum Inc, a VAR, MSP, and training provider based out of Chicago. He has over 20 years of experience focused on networking, security, and data center technologies. Tom is a Juniper Ambassador and is certified by Juniper as a JNCIE-ENT #424.

Pierre-Yves Maunier (Recipe 17) is a French Juniper Ambassador working as a Data Center and Network Engineering Director at Acorus Networks. For the past 15 years he's worked as a Network Architect for various types of companies : ISP, Hosting Provider, Content Network, and now Network Security dealing with DDoS mitigation systems. For each of these jobs he was working on Juniper based networks. His main focus, and what he loves, is to work in the core backbones and network interconnections that are the foundations of Internet.

Yasmin Lara (Recipe 13) is a Juniper Ambassador and a Juniper Networks certified instructor working at Sunset Learning Institute, a JNAEP based in Reston, VA. She teaches courses from the SP, ENT, SEC, Cloud, and DevOps tracks. She has several years of networking experience and worked for Juniper Networks as a Resident Engineer for a large service provider in the US, before joining SLI. Her biggest professional passion is to understand and learn about technology and figure out the best way to explain it to others. She is baseball and starwars fanatic and enjoys expending time with her husband and two teenage boys, and practicing taekwondo.

Stefan Fouant (Recipe 15) is the Director of Technology and Cloud Strategy at Sun Management with decades 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 is a quadruple JNCIE and also the author of *Day One: Junos Fusion Data Center Up and Running*.

Steve Puluka (Technical Editor) is a Network Architect 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 Pennsylvania. 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.

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.

Foreword

We in the networking community love to share experiences and to help others, as others have helped us throughout our careers. The Juniper Ambassador program identifies and highlights network engineers who have both a high-level of skill and a willingness to share their expertise with the community. This particular group of Juniper Networking engineers is from all across the globe, and they are an extremely passionate and caring team.

So welcome to the newest eclectic collection of sample configurations using Junos in your network. These annual Cookbook publications are a tangible manifestation of the Juniper Ambassadors sharing their latest experience in deploying and managing Juniper Networks technology of all kinds.

Contributing to a technical book is a labor of love. Network recipes require lots of time planning and outlining the technical problem and solution. And the recipes typically come out of the real-life work experiences that the Ambassadors have been facing as working engineers. So, the recipes have to be abstracted from our particular networks and recreated in the lab so they can be universally understood and shared outside the context that gave them birth. And finally, the lab configurations and data need to be logically organized and presented so they can be fruitfully used by others.

The recipes in these cookbooks stand the test of time in an industry notable for rapid change. That is because they are born out of the practical experiences of working engineers with technical reviews by fellow working Ambassadors. These recipes offer solutions to real problems in the field that are missing full explanations in the documentation, but they also exemplify the network thinking process on how to solve problems in your own network.

The Juniper Ambassadors and I hope that these pages save you both time and effort as working engineers, and that you're able to deploy their solutions into your own networks.

Steve Puluka, IP Architect, DQE Communications
& Juniper Ambassador
April 2019

Recipe 1: Virtualizing Routers with Routing Instances

By Martin Brown

- Junos OS Used: 12.3X48-D70.3
- Juniper Platforms General Applicability: SRX Series Firewall

Virtualization is commonplace in today's modern networks because it allows companies to share a physical device with multiple logical devices, using otherwise unused processing power and memory while also saving money, space, and power consumption. Usually virtualization is associated with servers, however, the Junos OS also allows you to create multiple virtual devices, such as a virtual switches or virtual routers on a single MX Series router, SRX Series firewall, or EX Series switch, often without any additional cost or licenses.

Problem

You work for a building management company. One of the buildings, 32 The Plaza, is a multi-tenancy office building currently occupied by two medium sized companies, ACME and Globomatics. Both of these companies require Internet access, however, you only have a single link to the Internet and a single SRX firewall. Your task is to allow both companies access to the Internet and allocate them IP addresses dynamically, not only ensuring inside devices are secured against outside attacks from the Internet, but also making sure devices on either network are unable to communicate directly with devices on the other network. This sounds like a perfect scenario to split the SRX into virtual routers. This sounds like a job for "routing instances."

Solution

The firewall is an SRX-110, which is currently using its default configuration. This firewall comes with a VDSL interface, however, the ISP has provided a standard Ethernet connection, so as illustrated in Figure 1.1, the connection to the ISP will be made via interface fe-0/0/0. In addition, one interface, fe-0/0/1, will be connected to a switch solely for ACME's use and another interface, fe-0/0/4, will be connected to Globomatics' switch.

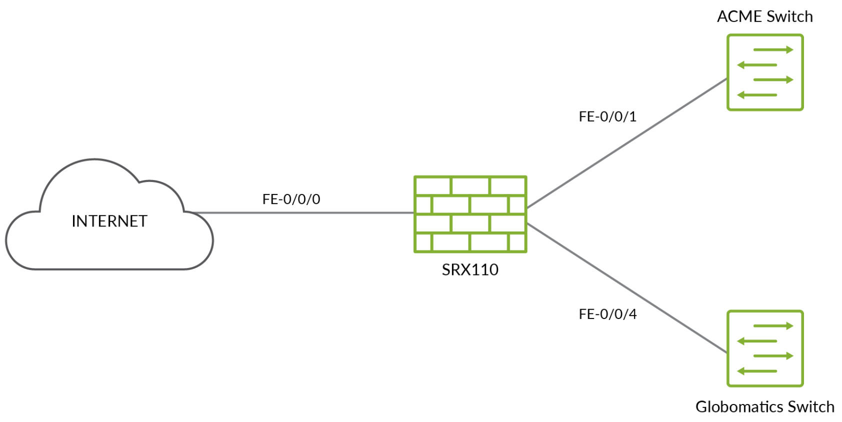


Figure 1.1 Physical Network Diagram at 32 The Plaza

The first task is to configure the master, or default instance. This is quite a straightforward task as it involves setting the hostname, configuring the IP address of the external interface, and creating a default route. The ISP has assigned a static IP address for our SRX: 192.0.2.10. The addresses used for each interface, both inside and out, are shown in Figure 1.2.

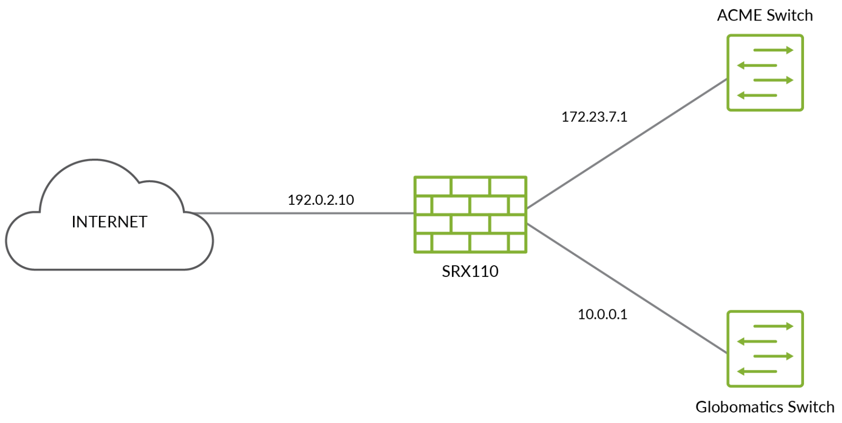


Figure 1.2 IP Address Assignments at 32 The Plaza

By default, all interfaces are part of the VLAN *vlan-trust*, therefore before you can assign an IP address, you must delete the family `ethernet-switching` first. Therefore, to delete the VLAN association and set the IP address on interface fe-0/0/0, you need to run the following:

```
delete interfaces fe-0/0/0 unit 0 family ethernet-switching
set interfaces fe-0/0/0 unit 0 family inet address 192.0.2.10/24
```

The default configuration creates a routed VLAN interface, vlan.0, and assigns an IP address of 192.168.1.1/24. This will be left in place as you can use it to manage the device. Because creating the routing instances won't clear the interface address configuration, the IP addresses for interfaces fe-0/0/1 and fe-0/0/4 can be created now, after first clearing the VLAN association, of course:

```
delete interfaces fe-0/0/1 unit 0 family ethernet-switching
set interfaces fe-0/0/1 unit 0 family inet address 172.23.7.1/24

delete interfaces fe-0/0/4 unit 0 family ethernet-switching
set interfaces fe-0/0/4 unit 0 family inet address 10.0.0.1/24
```

The hostname should now be set, 32-THE-PLAZA, and the default static route should also be set. The next hop address is 192.0.2.254:

```
Set system host-name 32-THE-PLAZA
set routing-options static route 0.0.0.0/0 next-hop 192.168.1.254
```

IMPORTANT Before the configuration can be committed, the command `set system root-authentication plain-text-password` must be run, otherwise the commit will fail.

In theory, while the devices in the management network should have Internet connectivity through the SRX, the management network would not be secure because the interface connected to the Internet is not part of a security zone. In practice, however, if interfaces are not part of a zone, the SRX will not allow traffic to pass to and from that interface.

The default configuration of an SRX 110 creates two zones, `trust` and `untrust`, neither of which are descriptive enough for this solution, therefore the default zones will be deleted and a new outside zone called `INTERNET` will be created, to which the interface fe-0/0/0.0 will be assigned and a new inside zone will be created and given the name `MANAGEMENT-TRUST`. The interface vlan.0 will be added to this inside zone:

```
delete security zone security-zone untrust
delete security zone security-zone trust

set security zones security-zone INTERNET interfaces fe-0/0/0.0
set security zones security-zone MANAGEMENT-TRUST interfaces vlan.0
```

The SRX must also be told which services, such as DHCP discover messages or ICMP requests, to allow into the zone. In our case, all services and protocols will be allowed into the zone `MANAGEMENT-TRUST`:

```
set security zones security-zone MANAGEMENT-TRUST host-inbound-traffic system-services all
set security zones security-zone MANAGEMENT-TRUST host-inbound-traffic protocols all
```

SRX devices have a very useful security policy called a *screen*. This protects the device against attack attempts. The policy is configured as follows:

```
security {
    screen {
        ids-option untrust-screen {
            icmp {
                ping-death;
            }
            ip {
                source-route-option;
                tear-drop;
            }
            tcp {
                syn-flood {
                    alarm-threshold 1024;
                    attack-threshold 200;
                    source-threshold 1024;
                    destination-threshold 2048;
                    timeout 20;
                }
                land;
            }
        }
    }
```

As you can see, the policy has been given the name `untrust-screen`. This name was used because the zone that previously used this policy was called the *untrust zone*. The default name can be left, however, it's better to rename it to make it more relevant. The name therefore will be renamed to `INTERNET-SCREEN` and assigned to the zone `INTERNET`. You can achieve both of these actions by using the following commands:

```
edit security screen
rename ids-option untrusted-screen to INTERNET-SCREEN
top

set security zones security-zone INTERNET screen INTERNET-SCREEN
```

Next, a security policy should be created to cover traffic going from the `MANAGEMENT-TRUST` zone to the `INTERNET` zone. This traffic should allow all traffic out. In addition, the default policy should be deleted in order to keep the configuration clean and to reduce the chances of any errors occurring during a commit:

```
delete security policies

edit security policies from-zone MANAGEMENT-TRUST to-zone INTERNET
set policy MANAGEMENT-OUT match source-address any destination-address any application any
set policy MANAGEMENT-OUT then permit
```

Because the internal interfaces are using RFC1918 addresses, these addresses cannot be used as source or destination addresses over the Internet, so a NAT policy must be created. First, the default policy should be deleted, then a source NAT policy will be created along with a rule set called `MANAGEMENT-TO-INTERNET`. The source zone will be MANAGEMENT-TRUST and the destination zone will be INTERNET. The rule set will be told to match against any IP address therefore the source address will be set as 0.0.0.0/0:

```
delete security nat

set security nat source rule-set MANAGEMENT-TO-INTERNET from zone MANAGEMENT-TRUST
set security nat source rule-set MANAGEMENT-TO-INTERNET to zone INTERNET
set security nat source rule-set MANAGEMENT-TO-INTERNET rule MANAGEMENT-NAT match source-
address 0.0.0.0/0
set security nat source rule-set MANAGEMENT-TO-INTERNET rule MANAGEMENT-NAT then source-
nat interface
```

Finally, steps must be taken to ensure that personnel in ACME or Globomatics are unable to attempt access to the device via SSH, telnet, or a web browser. A firewall filter should be created to allow management access only from the subnet 192.168.1.0/24. All other traffic should be rejected:

```
set firewall family inet filter MANAGEMENT-ACCESS term ALLOWED from address 192.168.1.0/24
set firewall family inet filter MANAGEMENT-ACCESS term ALLOWED then accept
set firewall family inet filter MANAGEMENT-ACCESS term NOTALLOWED then reject
```

This firewall filter should be applied to the loopback 0 interface:

```
set interfaces lo0 unit 0 family inet filter input MANAGEMENT-ACCESS
```

Now that the management network has been created and secured, our next task is to create the instances. Two instances will be created, ACME and GLOBOMATICS. The interface fe-0/0/1.0 will be assigned to the instance ACME, whereas interface fe-0/0/4.0 will be assigned to the instance GLOBOMATICS. Both instance types will be set as “virtual-router”. This means separate routing tables will be created for each routing instance:

```
set routing-instances ACME instance-type virtual-router
set routing-instances ACME interface fe-0/0/1.0

set routing-instances GLOBOMATICS instance-type virtual-router
set routing-instances GLOBOMATICS interface fe-0/0/4.0
```

Now the issue we have is that there is only a single interface connected to the Internet. An interface can only be added to a single routing instance. If fe-0/0/0 was added to the routing instance ACME, neither the master instance nor GLOBOMATICS would be able to use the interface. The alternatives to adding individual interfaces to each instance is to *leak* interfaces between routing instances or to leak routes. *Leaking* is where Junos OS copies a component of the routing table of one instance to the routing table of another instance.

If interfaces are leaked, all interfaces of that instance will be copied to the other routing instance, including the inside interface, which could be a security risk, therefore in this case, the routes should be leaked. While Junos OS allows leaking of routes from dynamic routing protocols, such as BGP or OSPF, this SRX has a single static route in the master instance, therefore this static default route will be shared with ACME and GLOBOMATICS.

To do this, a routing information base (RIB), also known as routing table, needs to be created, and this will then be set to import a routing table, or RIB from another routing instance and then specify which RIB the route is going from and to. In this case, a RIB group called MASTER-TO-INSTANCES will be created and it will tell Junos OS to import the RIB to ACME.inet.0 and GLOBOMATICS.inet.0 from inet.0. If square brackets are included several RIBs can be entered at once:

```
set routing-options rib-groups MASTER-TO-INSTANCES import-rib [ inet.0 ACME.inet.0 GLOBOMATICS.
inet.0 ]
```

Junos must then be told where to apply this RIB group. In this case, the RIB group will be applied to static routes under routing-options for the master routing instance:

```
set routing-options static rib-group MASTER-TO-INSTANCES
```

By creating these routing instances, our logical topology has changed and it now looks similar to that shown in Figure 1.3 where, logically, there are now three SRX firewalls.

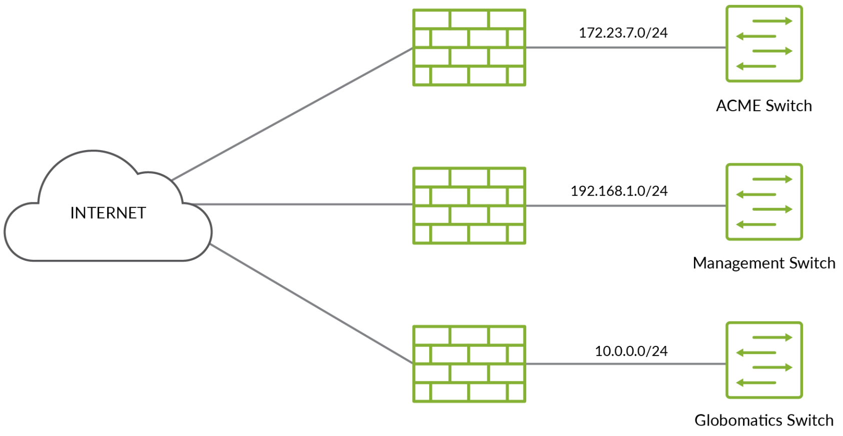


Figure 1.3 Logical Network at 32 The Plaza

Given that there are now ‘three’ firewalls at the Internet edge, we really should consider securing the two new firewalls just as was done with the master instance. The first thing would be to create the new zones. These will be called ACME-TRUST and GLOBOMATICS-TRUST and in this case, all protocols and system services will be allowed:

```
edit security zones security-zone ACME-TRUST
set interfaces fe-0/0/1.0 host-inbound-traffic system-services all
set interfaces fe-0/0/1.0 host-inbound-traffic protocols all
```

```
up
edit security-zone GLOBOMATICS-TRUST
set interfaces fe-0/0/4.0 host-inbound-traffic system-services all
set interfaces fe-0/0/4.0 host-inbound-traffic protocols all
top
```

NOTE The zones are created under the global configuration as opposed to being created under the routing instance configuration. This is the same for all security policies and NAT rule sets.

Once the zones have been created, the policies to allow traffic out from the zone to the Internet must be configured. In this case, all traffic will be allowed out from each zone:

```
edit security policies from-zone ACME-TRUST to-zone INTERNET
set policy ACME-OUT match source-address any
set policy ACME-OUT match destination-address any
set policy ACME-OUT match application any
set policy ACME-OUT then permit
top

edit security policies from-zone GLOBOMATICS-TRUST to-zone INTERNET
set policy GLOBOMATICS-OUT match source-address any
set policy GLOBOMATICS-OUT match destination-address any
set policy GLOBOMATICS-OUT match application any
set policy GLOBOMATICS-OUT then permit
top
```

Finally, in order to allow traffic to reach the Internet, NAT policies, and rule sets should be created. In this case, all traffic from that zone will be translated to the interface address of ge-0/0/0.0. The first rule set will be from the zone `ACME-TRUST` to the `INTERNET` zone:

```
set security nat source rule-set ACME-TO-INTERNET from zone ACME-TRUST
set security nat source rule-set ACME-TO-INTERNET to zone INTERNET
set security nat source rule-set ACME-TO-INTERNET rule ACME-NAT match source-address 0.0.0.0/0
set security nat source rule-set ACME-TO-INTERNET rule ACME-NAT then source-nat interface
```

The second rule set will be from the zone `GLOBOMATICS-TRUST` to the zone `INTERNET`:

```
set security nat source rule-set GLOBOMATICS-TO-INTERNET from zone GLOBOMATICS-TRUST
set security nat source rule-set GLOBOMATICS-TO-INTERNET to zone INTERNET
set security nat source rule-set GLOBOMATICS-TO-INTERNET rule GLOBOMATICS-NAT match source-
address 0.0.0.0/0
set security nat source rule-set GLOBOMATICS-TO-INTERNET rule GLOBOMATICS-NAT then source-
nat interface
```

Traffic should now be allowed to flow from all three routing instances, out to the Internet, and as the SRX is a stateful firewall, it should be allowed to return. There is still one more task to perform and that's configuring the SRX as a DHCP server.

Adding a DHCP server to routing instances is performed under the routing instance itself. Typically, DHCP servers are configured under `system | services | DHCP`, however, this hierarchy level is not available under routing instances. Instead, these have to be configured under `system | services | dhcp-local-server`.

First, a group must be created and assigned to an interface. For ACME, a group ACME-POOL will be created. This will be assigned to the inside interface, which is fe-0/0/1.0. For GLOBOMATICS, the group GLOBOMATICS-POOL will be created and assigned to interface fe-0/0/4.0:

```
edit routing-instances ACME
set system services dhcp-local-server group ACME-POOL interface fe-0/0/1.0
top

edit routing-instances GLOBOMATICS
set system services dhcp-local-server group GLOBOMATICS-POOL interface fe-0/0/4.0
```

Next, the pool itself needs to be created. This should include the network address this pool belongs to, the range of addresses available to the pool, the lease time, which is set as DHCP attributes along with the default gateway, the domain name, and the DNS servers, or *name* servers.

In this situation, ACME would like a domain name of acme.com to be set. The lease time should be 8 hours and the DNS servers should be 4.2.2.2 and 8.8.8.8. The default gateway will be the SRX's interface address.

The pool is configured under the access | address-assignment hierarchy:

```
edit routing-instances ACME access address-assignment
set pool ACME-POOL family inet network 172.23.7.0/24
set pool ACME-POOL family inet range ACME low 172.23.7.10
set pool ACME-POOL family inet range ACME high 172.23.7.250
set pool ACME-POOL family inet dhcp-attributes maximum-lease-time 28800
set pool ACME-POOL family inet dhcp-attributes domain-name acme.com
set pool ACME-POOL family inet dhcp-attributes name-server 4.2.2.2
set pool ACME-POOL family inet dhcp-attributes name-server 8.8.8.8
set pool ACME-POOL family inet dhcp-attributes router 172.23.7.1
top
```

Globomatics requires a similar configuration. Their network is obviously 10.0.0.0/24, however, they would like to use the OpenDNS DNS servers for clients, which are IP addresses 208.67.222.222 and 208.67.220.220:

```
edit routing-instances GLOBOMATICS access address-assignment
set pool GLOBOMATICS-POOL family inet network 10.0.0.1/24
set pool GLOBOMATICS-POOL family inet range GLOBOMATICS low 10.0.0.10
set pool GLOBOMATICS-POOL family inet range GLOBOMATICS high 10.0.0.250
set pool GLOBOMATICS-POOL family inet dhcp-attributes maximum-lease-time 28800
set pool GLOBOMATICS-POOL family inet dhcp-attributes domain-name globo.com
set pool GLOBOMATICS-POOL family inet dhcp-attributes name-server 208.67.222.222
set pool GLOBOMATICS-POOL family inet dhcp-attributes name-server 208.67.220.220
set pool GLOBOMATICS-POOL family inet dhcp-attributes router 10.0.0.1
```

These DHCP pools have now been created as per customer requirements, however, there is a problem. The SRX creates a DHCP pool for vlan.0 by default, but if you look at the configuration for this, you'll see the following error:

```
dhcp {
##
## Warning: Incompatible with 'system services dhcp-local-server group'
##

        pool 192.168.1.0/24 {
            address-range low 192.168.1.2 high 192.168.1.254;
            router {
                192.168.1.1;
            }
        }
        propagate-settings fe-0/0/0.0;
    }
```

This means you cannot use the traditional way of configuring a DHCP server on an SRX if you have created a DHCP pool using this new method. The default DHCP pool must be deleted:

```
edit system services
delete dhcp
top
```

As this DHCP pool was being used by the `MANAGEMENT` instance, a new DHCP group must be created and assigned to vlan.0:

```
set system services dhcp-local-server group MANAGEMENT interface vlan.0
```

The pool itself must then be created and in this case, the domain will be *theplaza.com* and the name server will be 192.168.1.254:

```
edit access address-assignment
set pool MANAGEMENT family inet network 192.168.1.0/24
set pool MANAGEMENT family inet range MANAGEMENT low 192.168.1.10
set pool MANAGEMENT family inet range MANAGEMENT high 192.168.1.250
set pool MANAGEMENT family inet dhcp-attributes maximum-lease-time 28800
set pool MANAGEMENT family inet dhcp-attributes domain-name theplaza.com
set pool MANAGEMENT family inet dhcp-attributes name-server 192.168.1.254
set pool MANAGEMENT family inet dhcp-attributes router 192.168.1.1
```

Once you have committed the configuration, clients should be able to perform a DHCP discover and receive an IP address. To check whether clients are broadcasting discovers, run the `show dhcp server statistics` command. As routing instances have been configured, the option `routing-instance <instance-name>` should also be included:

```
admin@32-THE-PLAZA> show dhcp server statistics routing-instance ACME
Packets dropped:
    Total                     0

Messages received:
    BOOTREQUEST               0
    DHCPDECLINE               0
    DHCPDISCOVER              0
    DHCPINFORM                0
    DHCPRELEASE               0
    DHCPREQUEST               0
```

```
Messages sent:
    BOOTREPLY                 0
    DHCPOFFER                 0
    DHCPACK                   0
    DHCPNAK                   0
    DHCPFORCERENEW            0
```

Unfortunately, as you can see, the statistics are all showing a count of 0. This means the SRX is not seeing any discover messages. The simple reason for this is the firewall filter that was created earlier allows management access from 192.168.1.0/24, but denies all other traffic including ICMP requests, dynamic routing protocols, and more importantly in our case, DHCP discover messages. You therefore need to inset a `term` into the filter to permit DHCP traffic:

```
edit firewall family inet filter MANAGEMENT-ACCESS
set term DHCP from destination-port dhcp
set term DHCP then accept
```

This `term` must then be inserted before the `NOTALLOWED` term:

```
insert term DHCP before term NOTALLOWED
```

Clients should now be assigned IP addresses from the SRX DHCP service, so let's verify that everything is now working as expected.

Verification

The first thing that should be checked is whether DHCP clients are being sent DHCP offers and that they are requesting these offers. The command `show dhcp server binding` will display all DHCP requests that have been acknowledged and as such are now associated with a MAC address. As the routing instances are running a separate DHCP server, the option `routing-instance <instance-name>` must be included at the end of the `show` command:

```
admin@32-THE-PLAZA> show dhcp server binding routing-instance ACME

IP address         Session Id  Hardware address   Expires     State      Interface
172.23.7.10        1           38:c9:86:09:3c:9f  28575       BOUND      fe-0/0/1.0
```

As you can see, the address 172.23.7.10 has been bound to the MAC address 38:c9:86:09:3c:9f proving that the DHCP server is offering addresses.

The next thing to check is whether the default static route is being leaked into the routing instances. The `show route` command will display all routing tables for all instances. The first table is inet.0, which belongs to the master instance where the default static route originates, and you should check that the route is here first, or else there won't be any routes to leak to the other tables:

```
admin@32-THE-PLAZA> show route

inet.0: 6 destinations, 6 routes (6 active, 0 holddown, 0 hidden)
```

```
+ = Active Route, - = Last Active, * = Both

0.0.0.0/0          *[Static/5] 00:21:09
                    > to 192.0.2.254 via fe-0/0/0.0
192.168.1.0/24     *[Direct/0] 00:00:27
                    > via vlan.0
192.168.1.1/32     *[Local/0] 00:21:21
                      Local via vlan.0
192.168.1.10/32    *[Access-internal/12] 00:00:20
                    > to 192.168.1.1 via vlan.0
192.0.2.0/24     *[Direct/0] 00:21:09
                    > via fe-0/0/0.0
192.0.2.10/32    *[Local/0] 00:21:12
                      Local via fe-0/0/0.0
```

As is evident, the default static route is present in inet.0, so the next table to look at belongs to the instance `ACME`. This is named `ACME.inet.0` and it appears just below inet.0:

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

0.0.0.0/0          *[Static/5] 00:40:21
                    > to 192.168.1.254 via fe-0/0/0.0
172.23.7.0/24      *[Direct/0] 00:00:10
                    > via fe-0/0/1.0
172.23.7.1/32      *[Local/0] 00:40:24
                      Local via fe-0/0/1.0
172.23.7.11/32     *[Access-internal/12] 00:00:07
                    > to 172.23.7.1 via fe-0/0/1.0
```

The default static route appears in ACME.inet.0 too, which is great news. Interestingly, the next hop is via fe-0/0/0.0, which isn't part of the ACME routing instance but it still appears as a valid interface simply because of the leaking from the master instance.

Just below ACME.inet.0 is `GLOBOMATICS.inet.0`, which is the routing table for the GLOBOMATICS instance:

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

0.0.0.0/0          *[Static/5] 00:39:34
                    > to 192.168.1.254 via fe-0/0/0.0
10.0.0.0/24        *[Direct/0] 00:00:08
                    > via fe-0/0/4.0
10.0.0.1/32        *[Local/0] 00:39:37
                      Local via fe-0/0/4.0
10.0.0.10/32       *[Access-internal/12] 00:00:04
                    > to 10.0.0.1 via fe-0/0/4.0
```

The default static route also appears in GLOBOMATICS.inet.0 too. This proves that the route leaking is working as expected.

Now that we know the leaking is working, the next task is to check whether the SRX is performing Network Address Translations (NATs), the command `show security nat source rule <rule-name>` will display the source address to be translated, the action the NAT rule should take, and the translation statistics such as the number of successful or failed hits and the number of sessions. In this case, the rule "`ACME-NAT`" will be checked:

```
admin@32-THE-PLAZA> show security nat source rule ACME-NAT
source NAT rule: ACME-NAT               Rule-set: ACME-TO-INTERNET
  Rule-Id                    : 1
  Rule position              : 1
  From zone                  : ACME-TRUST
  To zone                    : INTERNET
  Match
    Source addresses         : 0.0.0.0         - 255.255.255.255
  Action                        : interface
    Persistent NAT type         : N/A
    Persistent NAT mapping type : address-port-mapping
    Inactivity timeout          : 0
    Max session number          : 0
  Translation hits           : 176
    Successful sessions      : 176
    Failed sessions          : 0
  Number of sessions         : 10
```

The output in this case shows that translation is to be performed for addresses between 0.0.0.0 and 255.255.255.255, which is all IP addresses. The action is `interface`, which means it translates it to the interface address. Finally, there have been 176 hits, all of which were successful.

Our last check is to ensure that the inside clients can access resources on the Internet, and more importantly, that the traffic is going via the correct path. Traceroute is the perfect tool for this. If this is run from a client in ACME's network, this would prove everything is working as expected:

```
Katie:~ martinbrown$ traceroute 8.8.8.8
traceroute to 8.8.8.8 (8.8.8.8), 64 hops max, 52 byte packets
  1 172.23.7.1 0 msec 0 msec 8 msec
  2 192.0.2.254 9 msec 8 msec 9 msec
…
<output snipped for brevity>
…
 12 google-public-dns-a.google.com (8.8.8.8) 17 msec 16 msec 17 msec
```

Traceroute is showing that the traffic is first going via 172.23.7.1, then it's being forwarded to the next hop of 192.0.2.254 before eventually reaching the destination of 8.8.8.8, proving that traffic flow is exactly as per requirements, delivering secure Internet access to multiple customers, and using a single firewall, which in turn is using a single Internet connection.

Discussion

There are a number of reasons for wanting to use routing instances. Examples include creating an instance for site-to-site VPNs while using another instance for normal Internet traffic or having an instance just for guest Internet access. In the scenario used in this recipe, the instances were created for a multi-tenancy building. Whatever your reason for wanting to create them, as long as you follow each step carefully, routing instances aren't too difficult to configure and will allow you to fully utilize the power of your Juniper SRX Series firewall; that is, if you aren't using it to its maximum potential already.

Recipe 2: Saving Time with Apply-stuff

By Michel Tepper

- Junos OS Used: 18.1R2
- Juniper Platforms General Applicability: General

Junos is all about efficient, easy-to read configurations. In this recipe you will see how this can be taken to a higher level with apply-groups and apply-paths. These techniques are often seen in service provider configurations, but less so in enterprise or security configs. If you work with Junos, but never worked with apply-groups and/or apply-paths, it certainly is worth your time to keep reading!

Problem

On some configurations you need to specify the same settings multiple times. You might be working on a switch with 48 1G ports and all of those ports need to be set to full-duplex, linkspeed 1G, and no-auto negotiate. Or you might want to set the host-inbound-traffic on all your interfaces but some with addition, so you can't use zone-level settings. Surely there must be an efficient way to this.

And then there is this problem: You want to filter certain traffic with stateless firewall filters on ingress interfaces. You know you can use prefix lists to build neat policy. But it's annoying that you have to change the prefix list when you change the IP address of an interface. Surely there's a way around this?

Solution

Apply groups.

Let's take a look at the first problem: the same setting multiple times in the configuration, and then go back to the problem described: fixing 48 interface settings. You could of course copy the settings 40+ times or use the "wildcard set" statement. That saves time during the configuration but still leaves you with 48 times the same setting in the configuration. Let's use `apply-groups` instead.

What is an apply group? Basically, you could call it a macro with pattern matching. The use of apply groups requires two steps:

- Define the apply group with a pattern matching in it.
- Activate the apply group in the configuration.

Let's first define our `apply-group`. In this case we want a macro to replace the following interface settings:

```
root@fw1# show interfaces ge-0/0/0
speed 1g;
link-mode full-duplex;
ether-options {
    no-auto-negotiation;
}
```

But not only for ge-0/0/0, but for all 1G interfaces. So, in wildcard terms for ge-*, you can do this by defining a group:

```
set groups 1g-interfaces interfaces <ge-*> speed 1g
set groups 1g-interfaces interfaces <ge-*> link-mode full-duplex
set groups 1g-interfaces interfaces <ge-*> ether-options no-auto-negotiation
```

Note two things here. First a user-defined name (`1g-interfaces`) was given to the apply group. Second, the wildcard replaces the interface name placed between the brackets < >. The configuration looks like this:

```
root@fw1# show groups 1g-interfaces

interfaces {
    <ge-*> {
        speed 1g;
        link-mode full-duplex;
        ether-options {
            no-auto-negotiation;
        }
    }
}
```

Now let's move on to the next step and apply the group. This can be done at any level you see fit. You can apply all groups you write at the "root" level, or apply them at the level you have written them for so there's clarity when you read through the configuration. Let's apply the group `1g-interfaces` at the interface level:

```
set interfaces apply-groups 1g-interfaces
```

And `show interface ge-0/0/0`:

```
root@fw1# show interfaces ge-0/0/0
unit 0 {
    family inet {
        address 10.31.5.226/24;
    }
}
```

What? Nothing happened to ge-0/0/0? Well, something did happen, the `group` was applied. You just don't see it until you pipe the `show` command through the `display inheritance` filter:

```
root@fw1# show interfaces ge-0/0/0 | display inheritance
##
## '1g' was inherited from group '1g-interfaces'
##
speed 1g;
##
## 'full-duplex' was inherited from group '1g-interfaces'
##
link-mode full-duplex;
##
## 'ether-options' was inherited from group '1g-interfaces'
##
ether-options {
    ##
    ## 'no-auto-negotiation' was inherited from group '1g-interfaces'
    ##
    no-auto-negotiation;
}
unit 0 {
    family inet {
        address 10.31.5.226/24;
    }
}
```

Now you can see the setting from our apply group was assigned to ge-0/0/0 and to all other interfaces starting with ge-, so all gigabit interfaces.

What if one, or a few, interfaces didn't have this setting? Easily solved: use `apply-group-except` at that more specific level. Let's say interface ge-0/0/6 needs to `auto-negotiate`, so now it looks like this:

```
root@fw1# show ge-0/0/6 | display inheritance
##
## '1g' was inherited from group '1g-interfaces'
##
speed 1g;
##
## 'full-duplex' was inherited from group '1g-interfaces'
##
link-mode full-duplex;
##
## 'ether-options' was inherited from group '1g-interfaces'
##
ether-options {
    ##
    ## 'no-auto-negotiation' was inherited from group '1g-interfaces'
    ##
    no-auto-negotiation;
}
unit 0 {
    family inet {
        address 10.1.1.254/24;
    }
}
```

And configure the `apply-groups-except` on ge-0/0/6:

```
set ge-0/0/6 apply-groups-except 1g-interfaces
```

Checking the results shows:

```
root@fw1# show ge-0/0/6 | display inheritance
apply-groups-except 1g-interfaces;
unit 0 {
    family inet {
        address 10.1.1.254/24;
    }
}
```

So now the apply group is applied to all gigabit interfaces except ge-0/0/6. Exactly what we wanted.

Now let's try to solve the second question: *accept pings on all interfaces*. The definition of that apply group could look like this:

```
accept-ping {
    security {
        zones {
            security-zone <*> {
                interfaces {
                    <*> {
                        host-inbound-traffic {
                            system-services {
                                ping;
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Next, `apply` this to the security zones level:

```
[edit security zones]
root@fw1# set apply-groups accept-ping
```

Checking the result shows:

```
[edit security zones]
root@fw1# show
apply-groups accept-ping;


… … … … …
security-zone internal {
    interfaces {
        ge-0/0/1.0;
    }
}
```

Nothing at a first glance, but let's see what the display inheritance filter shows:

```
security-zone internal {
    interfaces {
        ge-0/0/1.0 {
            ##
            ## 'host-inbound-traffic' was inherited from group 'accept-ping'
            ##
            host-inbound-traffic {
                system-services {
                    ##
                    ## 'ping' was inherited from group 'accept-ping'
                    ##
                    ping;
                }
            }
        }
    }
}
```

And again, it's the result we wanted: all interfaces on all zones will get `host-inbound-traffic system-services` pings configured. Of course, you could use `apply-group-except` again on the zone.

Okay, time to move on to the next issue. To avoid an overloaded system with fake Internet Key Exchange (IKE) requests you only want to accept IKE requests on the incoming interface (ge-0/0/0) when it comes from known IKE peers. In a configuration it looks like this:

```
policy my_policy {
    proposal-set standard;
    pre-shared-key ascii-text "$9$j7iPQ/9pBRStuyKM8dVk.m536Ap0"; ## SECRET-DATA
}
gateway to_peer1 {
    ike-policy my_policy;
    address 1.1.1.1;
    external-interface ge-0/0/0;
}
gateway to_peer2 {
    ike-policy my_policy;
    address 2.2.2.2;
    external-interface ge-0/0/0;
}

prefix-list ike-peers {
    1.1.1.1/32;
    2.2.2.2/32;
}

 filter ike-filter {
    term 1 {
        from {
            source-prefix-list {
                ike-peers;
            }
            destination-port [ 500 4500 ];
        }
        then accept;
    }
    term 2 {
        from {
            destination-port [ 4500 500 ];
        }
        then {
            discard;
        }
    }
    term 3 {
        then accept;
    }
}

root@fw1# show interfaces ge-0/0/0
unit 0 {
    family inet {
        filter {
            input ike-filter;
        }
        address 10.31.5.226/24;
    }
}
```

It's a lot of configuration, but it basically results in accepting IKE traffic only from IP addresses listed in the `prefix-list ike-peers`. All fine, but when you change an IP address in the IKE config, you also need to change the address in the prefix list. For this kind of situation, `apply-path` comes to help. It can fill a prefix list from a certain hierarchy level in the config. For now, we want it filled with all IP addresses found under the `security ike gateway <name> address`:

```
root@fw1# set apply-path "security ike gateway <*> address <*>"
```

This results in:

```
root@fw1# show policy-options prefix-list ike-peers
apply-path "security ike gateway <*> address <*>";
```

The two wildcards (<*>) match any name for the gateway, and any configured IP address. But how do we know what's in the prefix list? Use `display inheritance` again!

```
root@fw1# show policy-options prefix-list ike-peers | display inheritance
##
## apply-path was expanded to:
##     1.1.1.1;
##     2.2.2.2;
##
apply-path "security ike gateway <*> address <*>";
```

It's clear the correct IP addresses are taken from the configuration and put into the `prefix-list` from this. When we change the address of an IKE peer the IKE filter is updated automatically!

IMPORTANT Please note that you can only do this for IKE when all gateways have fixed IP-addresses. Dynamic peers will be filtered out. But the configuration can be used for other filters – BGP peers often are filtered this way.

Discussion

In the solution to this recipe's problem we showed the use of `apply-groups` and `apply-path`. Both techniques can help you to simplify the configuration and make maintenance easier, reducing the chance of mistakes when altering configurations.

Recipe 3: Enabling the Inet.3 Table for BGP to Use Label-Switched Paths

By Chris Parker

- vSRX Version Used: 12.1X47-D15.4
- Juniper Platforms General Applicability: MX, SRX, QFX

If you're running an MPLS network, the Junos OS gives you a phenomenal amount of control over choosing which traffic goes out of your router as IPv4 or IPv6, and which traffic goes out via a label-switched path (LSP). The key to understanding how to manipulate the method your traffic chooses is a special routing table called *inet.3*.

This recipe shows you what the inet.3 routing table is, and how you can configure your router to use a label-switched path when sending traffic to prefixes learned by BGP.

In this author's experience from teaching the topic, a common cause of confusion around inet.3 comes from students not quite understanding how BGP resolves next hops. For that reason, this recipe first walks you through how BGP works in a network with no MPLS at all. Then, after that, the recipe configures two different kinds of label protocols, and we'll see how these protocols change the behavior of traffic.

Problem

You want to turn on MPLS so you can take advantage of MPLS applications like traffic engineering, Layer 3 VPNs, pseudowires, and BGP-free cores. But you're new to BGP, and you don't know how the Junos inet.3 table works. You'd like to understand so you can take advantage of this extended functionality.

Solution

Step 1: Configuring BGP and Understanding Recursive Next Hops

You might already know about the routing tables inet.0 and inet6.0. They're the default global routing tables for IPv4 and IPv6, respectively. By default, static and directly-connected routes all get put into this table, as do prefixes learned by interior gateway protocols like OSPF or IS-IS.

On the whole, when a router learns prefixes via a protocol like OSPF or IS-IS, the next hop to those IPs will be the router that's actually giving us the advertisement. For example, examine the topology in Figure 3.1.

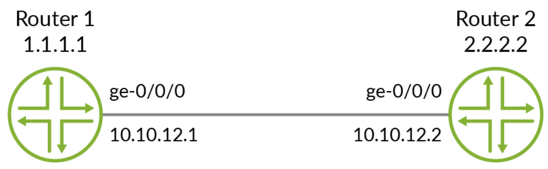


Figure 3.1 *A Simple Two-router Topology, Running Only IS-IS*

These two routers are running IS-IS. If Router 2 advertises prefixes to Router 1, Router 1's next hop will be 10.10.12.2, out of the directly connected interface going to Router 2. Let's look at how Router 1 thinks it gets to 2.2.2.2:

```
root@Router1> show route 2.2.2.2

inet.0: 13 destinations, 13 routes (12 active, 0 holddown, 1 hidden)
+ = Active Route, - = Last Active, * = Both

2.2.2.2/32         *[IS-IS/18] 00:37:31, metric 10
                    > to 10.10.12.2 via ge-0/0/0.0
```

Routes learned by BGP also go into the inet.0 table. However, next hops in BGP work very differently. You see, in an internal BGP network, every router in the network has to have a BGP peering with every single other router. The reason for this is that unlike OSPF, or IS-IS, BGP has no visibility of the topology, and therefore BGP has no way of spotting routing loops.

This is why prefixes learned by an iBGP neighbor aren't passed on to other iBGP neighbors: every router peers with every other router in the autonomous system, so there's no need to pass learned prefixes on to other peers in the same autonomous system. (You may have heard of something called a *route reflector* that helps us get around this, but let's put that aside for this recipe.)

What this means is that two routers at either end of a large network - in other words, two routers that are not directly connected to each other, that may have many routers in between them - will still have an iBGP peering and will use iBGP to advertise prefixes to each other.

This is an important concept: rather than advertising the prefixes to a directly-connected neighbor, and having that neighbor propagate the prefix throughout the network to its own neighbors, in iBGP an individual router will peer with every single other router in the network and advertise the prefixes to each of them individually.

So, when an iBGP neighbor isn't directly connected, what does the router use as the next hop? The answer is that it depends, and in a moment, we'll see an example of this. But in the spirit of not turning this single recipe into an entire book, for now let's just say that it's very common for network engineers to configure routers to actually advertise their own loopback as the next hop when they advertise prefixes. As a consequence, the router receiving the prefix has an extra job to do: if the next hop isn't directly connected, our plucky router has to do a second look up to work out the actual physical next hop.

In Figure 3.2, every router has an iBGP peering to every other router. In addition, Router 3 has an eBGP peering to Router 6.

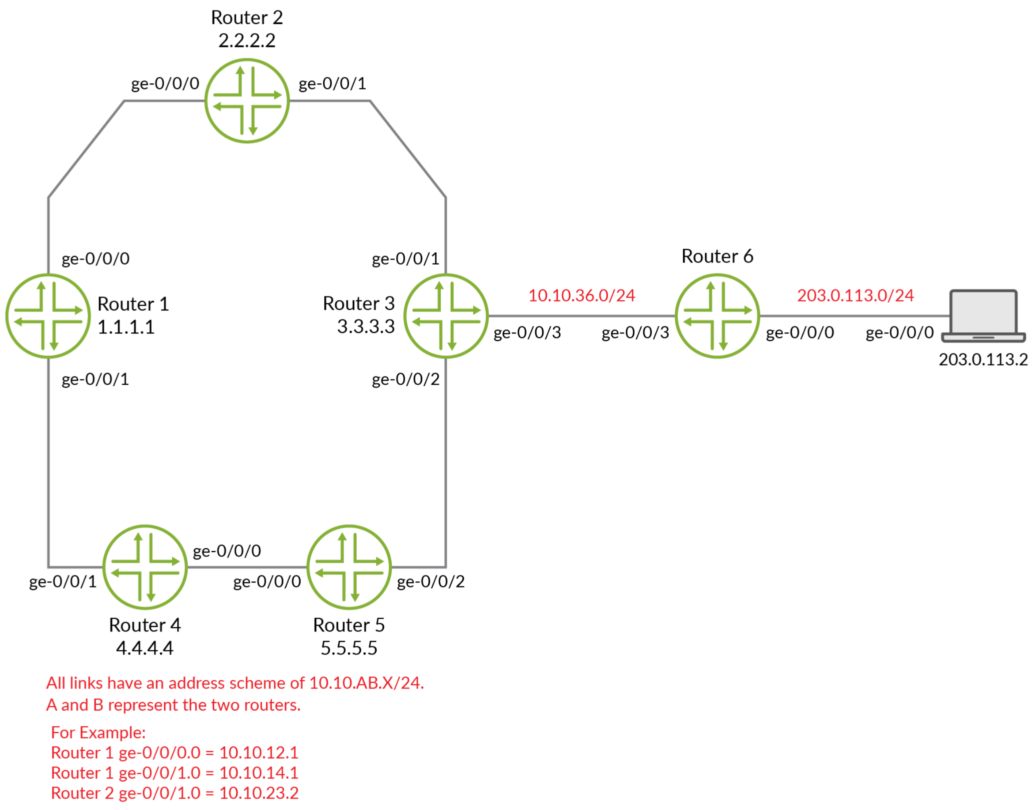


Figure 3.2 The Topology for Recipe 3

Every router in our AS64512 is set up in almost exactly the same way, so let's just focus on Router 1 for now. On every router in our AS we're going to turn on IS-IS, and in addition we're going to create an iBGP peering to every other router in the network:

```
set interfaces ge-0/0/0 unit 0 family iso
set interfaces ge-0/0/1 unit 0 family iso
set interfaces lo0 unit 0 family inet address 1.1.1.1/32
set interfaces lo0 unit 0 family iso address 49.0001.0000.0000.0001.00
set routing-options autonomous-system 64512
set protocols bgp group AS64512 type internal
set protocols bgp group AS64512 local-address 1.1.1.1
set protocols bgp group AS64512 neighbor 2.2.2.2
set protocols bgp group AS64512 neighbor 3.3.3.3
set protocols bgp group AS64512 neighbor 4.4.4.4
set protocols bgp group AS64512 neighbor 5.5.5.5
set protocols isis level 1 disable
set protocols isis level 2 wide-metrics-only
set protocols isis interface all
```

TIP In a lab it's easy and quick to switch on a protocol on all ports. But in the real world, you should be more selective, and only add a protocol to a port that requires it.

Here's the additional configuration on Router 3.

TIP It's best *not* to use an export policy like this in the real world.

```
set policy-options policy-statement REDISTRIBUTE_EVERYTHING_LOL from protocol direct
set policy-options policy-statement REDISTRIBUTE_EVERYTHING_LOL from protocol static
set policy-options policy-statement REDISTRIBUTE_EVERYTHING_LOL from protocol isis
set policy-options policy-statement REDISTRIBUTE_EVERYTHING_LOL then accept
set protocols bgp group TO_AS64513 type external
set protocols bgp group TO_AS64513 export REDISTRIBUTE_EVERYTHING_LOL
set protocols bgp group TO_AS64513 neighbor 10.10.36.6 peer-as 64513
```

In our beautiful lab, Router 6 is teaching Router 3 one prefix: 203.0.113.0/24. (Fun fact: this range is reserved by the IETF for documentation.)

If you look in the routing table of Router 3, you can see that 203.0.113.0/24 has a next hop of Router 6's physical interface, 10.10.36.6. This is exactly the same kind of result as we saw in the example at the start of this recipe: the next-hop for this prefix is the IP address of a directly-connected router. So far so good!

```
=root@Router3> show route 203.0.113.0/24

inet.0: 15 destinations, 16 routes (15 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

203.0.113.0/24     *[BGP/170] 00:04:54, localpref 100
                      AS path: 64513 I
                    > to 10.10.36.6 via ge-0/0/3.0
```

Technically, though, this link is outside of AS64512. We're not running IS-IS between R3 and R6. As such, the other routers in AS64512 don't know how to get to 10.10.36.6. And that's a shame – because by default, Router 3 doesn't actually change the next hop. If a router learns a prefix by eBGP, and advertises it to iBGP neighbors, by default the next hop is unchanged.

So Router 3 advertises 10.10.36.6 as the next hop to all the other routers in our network. If you look on Router 1, you'll see that this prefix isn't installed into the routing table, because R1 can't resolve the next hop – in other words, it doesn't know how to get to 10.10.36.6:

```
root@Router1> show route 203.0.113.0/24

inet.0: 13 destinations, 13 routes (12 active, 0 holddown, 1 hidden)


root@Router1> show route 203.0.113.0/24 hidden

inet.0: 13 destinations, 13 routes (12 active, 0 holddown, 1 hidden)
+ = Active Route, - = Last Active, * = Both

203.0.113.0/24      [BGP/170] 00:06:25, localpref 100, from 3.3.3.3
                      AS path: 64513 I
                      Unusable
```

This is why it's very common to add a `next-hop self` policy when you're taking prefixes from an eBGP peer and advertising them into iBGP. Let's add one to Router 3, so that when it advertises this prefix onto its internal peers, Router 3 changes the next hop to its own loopback, 3.3.3.3:

```
set policy-options policy-statement NEXT-HOP-SELF then next-hop self
set protocols bgp group AS64512 export NEXT-HOP-SELF
```

Let's look at some `extensive` output on R1 for this BGP-learned route 203.0.113.0/24. You'll now see two next hops: the `protocol next hop` (i.e. the next hop that we learned by BGP), and the actual next hop (i.e. how to get to that remote next hop). This is what is meant by a *recursive lookup*: we're resolving the protocol next hop, and then resolving the protocol next hop to find the actual physical next hop:

```
root@Router1> show route 203.0.113.0/24 extensive

inet.0: 13 destinations, 13 routes (13 active, 0 holddown, 0 hidden)
203.0.113.0/24 (1 entry, 1 announced)
TSI:
KRT in-kernel 203.0.113.0/24 -> {indirect(262142)}
        *BGP    Preference: 170/-101
                Next hop type: Indirect
                Address: 0x934c8c8
                Next-hop reference count: 3
                Source: 3.3.3.3
                Next hop type: Router, Next hop index: 557
                Next hop: 10.10.12.2 via ge-0/0/0.0, selected
                Protocol next hop: 3.3.3.3
                 {**snipped for brevity**}
```

So far, so good! IS-IS and BGP is running, everything is being learned automatically, and everything goes into the global routing table, inet.0. It's a tradition as old as time itself.

Now let's turn on MPLS – because when we do, the story is a little bit different. But to understand why, we first need to make sure we're "A-OK" with the concept of a label-switched path.

Step 2: Turning On MPLS and Understanding LSPs

An LSP is a path between two routers in a network that a packet will traverse to get to its destination. What makes this path different from a regular IPv4/IPv6 path is the use of labels to do the next hop lookup. All the routers in the path pre-agree that if a packet is destined for a certain destination, the sending router will add a label to the packet, to tell the receiving router what to do with it. No need to even look at the IPv4/IPv6 address!

If you've never done MPLS before, you might think: hang on a second - why would we want this? Don't our routers already know how to forward a packet? Isn't that the whole point of a router?

Well, yes, but when we create LSPs using MPLS, we get some very cool extra functionality beyond standard forwarding.

For example, suddenly you can run Layer 3 MPLS VPNs, because you can forward by label instead of by IP address, which gives you the ability to have the same private IP address space belong to multiple customers. Usually, if you used the same IP address on two interfaces, you'd get an error. MPLS VPNs frees us from these shackles.

You can perform "traffic engineering," where you create a path that goes a different way than what OSPF or IS-IS tells you is the best path, which allows you to perhaps send your priority traffic the *best* way, and send best-effort traffic a slightly longer way, so that you can make the best use of your available bandwidth.

You can even automatically calculate backup paths that kick in immediately if there's a problem in the network.

There's three ways we can create an MPLS LSP. The easy way is to turn on a protocol called LDP - the aptly named Label Distribution Protocol. By simply turning it on, your routers will form neighbor relationships with directly-connected routers, just like an IGP does, and automatically start advertising labels.

In fact, when you choose to use LDP, something interesting happens: Juniper routers *only* advertise labels for their loopback IP addresses. That means that if you turn on LDP in your example network, Router 1 will learn a label to get to 3.3.3.3, but it won't learn a label to get to 10.10.23.0/24, the network connecting Routers 2 and 3. In a moment, we'll find out why this is very useful indeed.

Let's add in LDP on all our routers. Once again, the configuration is essentially the same on each router, so let's concentrate on Router 1. We're going to turn on MPLS on all the relevant interfaces, and in addition we're also going to turn on LDP:

```
set interfaces ge-0/0/0 unit 0 family mpls
set interfaces ge-0/0/1 unit 0 family mpls
set protocols ldp interface all
set protocols mpls interface all
```

LDP is quick to set up, and it gives us applications like MPLS VPNs, but it doesn't give us any control: the traffic only ever follows the best path learned from our IGP.

That's where our second method comes in: a protocol called RSVP, or the Resource Reservation Protocol. In RSVP you have to manually create all of your LSPs, specifying them by name on every single router in the network. The advantage, though, is that you get precise control over exactly what path your LSP takes. You can specify that the path has to go via certain hops, you can specify certain links to avoid, and you can even automatically create new paths based on the bandwidth being used.

RSVP LSPs go in only one direction, so to make one path between two routers you need to configure both ends. As we did before, let's look at Router 1, where we're making an RSVP LSP to Router 3. For now, we'll just make a very simple path, that will also take the best path that IS-IS tells it to. However, there's plenty more we could add to this once it's up and running.

Weirdly, RSVP LSPs are configured under `protocols mpls`, not `protocols rsvp`:

```
set protocols rsvp interface all
set protocols mpls label-switched-path TO_ROUTER_3 to 3.3.3.3
```

NOTE If you're running IS-IS then you don't need to do anything else, but if you're running OSPF, aka the *coward's protocol*, you'll need to add one extra command: set protocols ospf traffic-engineering.

DOUBLE NOTE The above statement about OSPF being the *coward's protocol* does not reflect the opinion of Juniper in any way, is fully the comments of the author only, and is obviously only a joke. Having said that, it is also absolutely true.

This is a barebones RSVP configuration, meant to show you what it's like when it's up and running.

MORE? Another way to do this is to use the new protocol on the block: Segment Routing (SR) also known as SPRING. In short, SR provides extensions to OSPF and IS-IS so that you can create labeled paths without having to run an additional label distribution protocol, like LDP or RSVP. It's cool and new, and if you'd like to learn all about it Juniper has you covered: see Julian Lucek and Krzysztof Szarkowicz's excellent and, yes, free, *Day One: Configuring Segment Routing With Junos*, available at: https://www.juniper.net/us/en/training/jnbooks/day-one/configuring-segment-routing-junos/index.page.

Whatever protocol you choose to use, the information about these LSPs is stored in one of two places: mpls.0, or inet.3. We'll come back to mpls.0 later on. For now, let's look at inet.3.

Step 3: Verification via the Inet.3 Table

If a router is acting as the ingress for an label-switched path (or, to say it in English, if a router is acting as the start of the tunnel, where traffic enters, or ingresses, the LSP) then you'll find the IP address of the other end of the LSP (the "egress" of the tunnel) in the inet.3 table.

We'll explain how it's used in a moment. For now, let's look in the inet.3 table of Router 1. LDP is turned on in all interfaces on all routers, which means that you automatically have an LSP to the loopback of every router in the network. In addition, we've created one RSVP LSP, directly to Router 3's loopback. Let's take a look at what's there:

```
root@Router1> show route table inet.3

inet.3: 4 destinations, 5 routes (4 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

2.2.2.2/32         *[LDP/9] 00:11:21, metric 1
                    > to 10.10.12.2 via ge-0/0/0.0
3.3.3.3/32         *[RSVP/7/1] 00:00:16, metric 20
                    > to 10.10.12.2 via ge-0/0/0.0, label-switched-path TO_ROUTER_3
                    [LDP/9] 00:08:55, metric 1
                    > to 10.10.12.2 via ge-0/0/0.0, Push 299872
4.4.4.4/32         *[LDP/9] 00:09:27, metric 1
                    > to 10.10.14.4 via ge-0/0/1.0
5.5.5.5/32         *[LDP/9] 00:09:27, metric 1
                    > to 10.10.14.4 via ge-0/0/1.0, Push 299840
```

First of all, notice that by default, the *only* IPs in the inet.3 table are the loopback addresses of the other routers. *Why? In just a moment, we'll find out.*

Second, you can actually see two entries for 3.3.3.3: the LDP-signaled LSP, and the manually-created RSVP path, called "TO_ROUTER_3". You can see from the asterix * sign that the RSVP path is being chosen. RSVP has a route preference of 7, whereas LDP has 9, so RSVP paths will always get chosen over an LDP path by default.

So, now Router 1 knows about an LSP to the IP address 3.3.3.3. This means that if we were to ping 3.3.3.3 from Router 1, the outgoing packet would have a label attached to it, right? The traffic would go down the LSP, right? Right? Well… not quite.

You see, in Junos, the inet.3 table has one very specific job: *to resolve BGP next hops*.

The key thing to understand is that whenever a Juniper router learns a route by BGP, it will first try to resolve the protocol next hop – therefore, the BGP next hop – in the inet.3 table. If there isn't a match in inet.3, it doesn't matter: our router falls back to the inet.0 table instead. But by default, it looks in inet.3 first. And if it finds a match, it sends out traffic labeled.

In other words, traffic destined *to* 3.3.3.3 will *NOT* take the LSP. In fact, it will just go out of the router as a regular packet. But, if there's a prefix that:

- a) is in the regular inet.0 table,
- b) was learned via BGP, and,
- c) has a BGP next hop in the inet.3 table,

…then that traffic will indeed take the LSP.

That's a lot to take in, so once again let's refer to an example, by looking at what Router 1 sees now that MPLS has been added into the network. To save you from looking back, Figure 3.3 repeats the topology again.

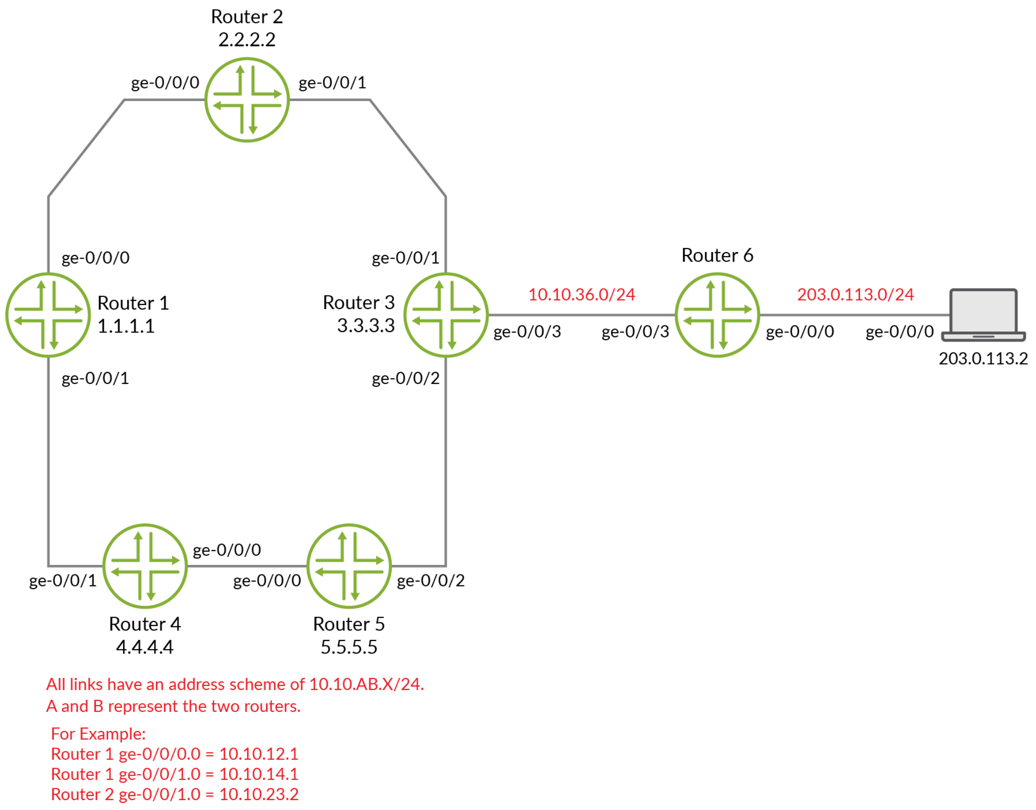


Figure 3.3 A Reminder of the Topology for Recipe 3

R1 is still learning about the 203.0.113.0/24 network, by BGP, from R3. Remember what happened earlier, with no MPLS. Let's imagine that Router 1 receives a packet destined to 203.0.113.2. R1 looks in inet.0, and sees it has a match for the

203.0.113.0/24 network, learned from 3.3.3.3. R1 then does a second look up, and discovers that it can get to 3.3.3.3 via 10.10.12.2. Finally, R1 pushes the packet out as a normal IP packet.

Now, let's run the same scenario, but with MPLS turned on. Suddenly, you'll see something different. Look at how the next hop has changed:

```
root@Router1> show route 203.0.113.0/24

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

203.0.113.0/24     *[BGP/170] 00:20:46, localpref 100, from 3.3.3.3
                      AS path: 64513 I
                    > to 10.10.12.2 via ge-0/0/0.0, label-switched-path TO_ROUTER_3
```

Router 1 once again looks up 203.0.113.0/24, and sees that it's learned about this prefix via BGP, from 3.3.3.3. But this time, Router 1 knows that it actually has an LSP to 3.3.3.3. It knows this because the first thing it did was to look in inet.3! As a result, R1 sends the packet out, but this time with a label saying that it's destined to 3.3.3.3. You can see this label by looking at the `extensive` output for the route:

```
root@Router1> show route 203.0.113.0/24 extensive

inet.0: 13 destinations, 13 routes (13 active, 0 holddown, 0 hidden)
203.0.113.0/24 (1 entry, 1 announced)
TSI:
KRT in-kernel 203.0.113.0/24 -> {indirect(262142)}
        *BGP    Preference: 170/-101
                Next hop type: Indirect
                Address: 0x934c8c8
                Next-hop reference count: 3
                Source: 3.3.3.3
                Next hop type: Router, Next hop index: 568
                Next hop: 10.10.12.2 via ge-0/0/0.0 weight 0x1, selected
                Label-switched-path TO_ROUTER_3
                Label operation: Push 299904
                Label TTL action: prop-ttl
                Protocol next hop: 3.3.3.3
                {snip...}
```

Where did this label 299904 come from? Well, R2 actually told R1 that this is the label it wants to receive, and knowing what R2 does with this label allows us to introduce the other routing table of interest to us: mpls.0.

The mpls.0 and inet.3 tables have similar but distinct tasks. In inet.3 you're going to find all the LSPs for which this router is acting as the "ingress" – in other words, the start of the tunnel. This is why BGP uses this table to resolve next hops. If there's a match in here, the router will know what label to add to the packet as it sends it on its merry way.

By contrast, our mpls.0 table is used when we receive a packet that already has a label on it. In other words, think of mpls.0 as the routing table for any LSPs where it's acting as a "transit" router, therefore a router in the middle of an LSP.

Let's have a look at what Router 2 does when it receives a packet with label 229004:

```
root@Router2> show route table mpls.0 label 299904

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

299904             *[RSVP/7/1] 00:06:44, metric 1
                    > to 10.10.23.3 via ge-0/0/1.0, label-switched-path TO_ROUTER_3
299904(S=0)        *[RSVP/7/1] 00:06:44, metric 1
                    > to 10.10.23.3 via ge-0/0/1.0, label-switched-path TO_ROUTER_3
```

Yes! It knows the name of the LSP configured on R1! That's the magic of RSVP, baby!

(*Fun bonus fact*: (S=0) refers to whether the "bottom-of-stack" bit has been set in the MPLS header. If it's S=1 the label is the bottom of the stack. If it's S=0 then there are one or more labels "underneath" this one, perhaps to tell the final destination router which VRF the traffic is part of.)

Discussion

So that's how you can take advantage of MPLS. If you just want the functionality of VPNs, use LDP. If you have an additional requirement for traffic engineering, use RSVP. And in any case, consider SR.

You may be wondering one final thing: *is it possible to send other traffic down label-switched paths, other than BGP-learned prefixes?* Of course, that it is indeed possible. In Recipe 4, we'll see how to do just that.

Recipe 4: Forcing Non-BGP Traffic to Take an LSP: Manipulating inet.3

By Chris Parker

- vSRX Version Used: 12.1X47-D15.4
- Juniper Platforms General Applicability: MX, SRX, QFX

In Recipe 3, you learned how the Junos OS will, by default, use the inet.3 table for one purpose only: to resolve BGP next hops.

This means that the only traffic that will go out as labeled MPLS traffic, and the only traffic that will traverse our labeled path, is traffic destined to prefixes learned by BGP, where the protocol next hop (therefore, the next hop that our BGP has advertised to us, as opposed to the actual next hop, which is the next physical router in the path) is known in the inet.3 table.

Okay, except what happens if the default behavior is no good for us? What if we actually want other traffic to also go down an LSP? Yes, most of the time the inet.3 table is only needed by BGP. But *most* of the time doesn't mean *all* of the time. Certain label protocols offer tremendous advantages when it comes to traffic engineering, such as instant fallback to pre-signaled backup paths, and so on. It sure would be nice if we could send other traffic down these paths as well.

Luckily, Junos makes it easy to manipulate this default behavior for your own needs. In fact, there's quite a few different ways to do it, and each method has its advantages and drawbacks. Some of these methods you'll regularly find use for in the real world and some of them are really just corner cases to fix specific occasional problems. This recipe looks at a few of the methods available to you, and discusses their pros and cons.

Problem

Currently your router takes all traffic in your BGP/MPLS network that's destined to a prefix learned by BGP and sends it out via an LSP. You want to know how to manipulate other traffic to also go down these paths, and to take advantage of the benefits MPLS brings, such as instant fallback to backup paths and traffic engineering.

Solution

Override 1: Install a Prefix Into Inet.3, and Associate It With an RSVP LSP

This recipe uses the same topology and configuration as Recipe 3.

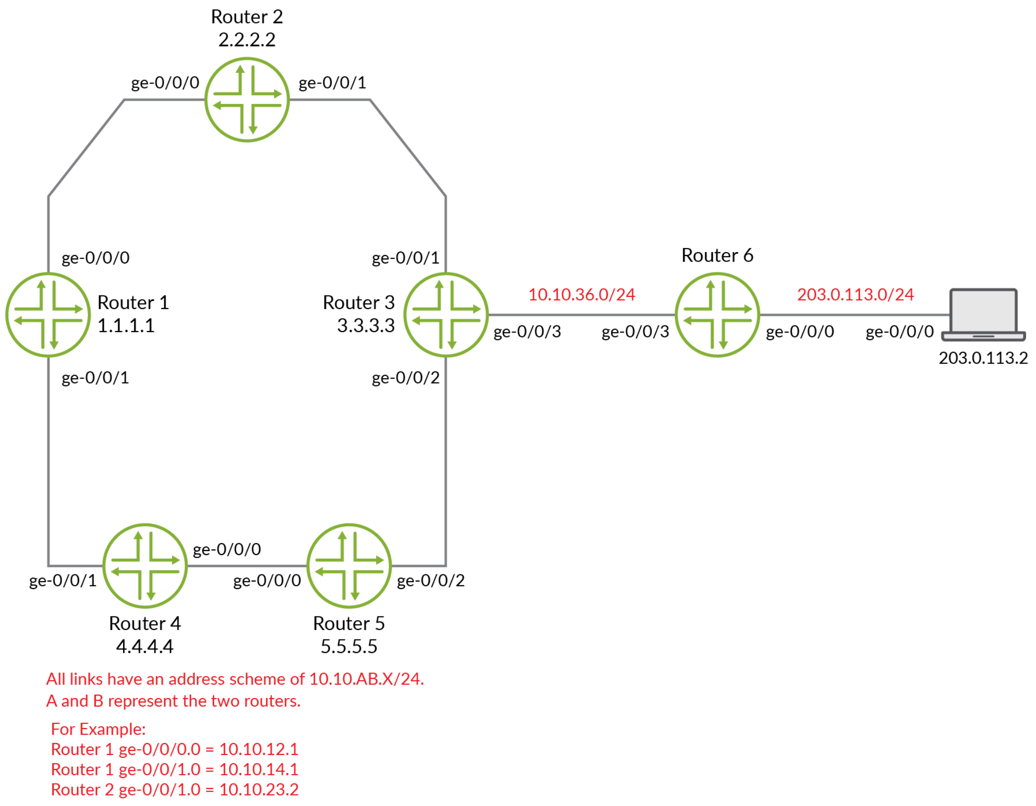


Figure 4.1 *The Topology for Recipe 4*

Just to remind you: Routers 1 to 5 are on network AS64512. Our routers are all running IS-IS to learn each other's links and loopbacks, and they're also all running BGP.

In addition, Router 3 is peering with a router outside of our network and learning 203.0.113.0/24. The link between Router 3 and 6 is outside of our network, and as such, none of the routers in AS64512 actually know how to get to 10.10.36.0.24.

Previously we had a policy on Router 3 that advertised its own loopback, 3.3.3.3, as the next hop to 203.0.113.0/24.

Let’s make two changes to this network. First, carrying on with the configuration as it stood at the end of Recipe 3, let’s remove Router 3’s NEXT-HOP-SELF policy, then redistribute this 10.10.36.0/24 link into our network. The configuration goes into Router 3:

```
delete protocols bgp group AS64512 export NEXT-HOP-SELF
set policy-options policy-statement EXPORT_CONNECTED from protocol direct
set policy-options policy-statement EXPORT_CONNECTED then accept
set protocols isis export EXPORT_CONNECTED
```

As far as Router 1 is concerned, if it wants to get to 203.0.113.0/24, its next hop is 10.10.36.6 – the interface IP of Router 6. Remember, when Router 3 learns about a pre x from an eBGP peer, and then advertises that pre x on to its iBGP peers (therefore, its internal neighbors), it doesn’t change the next hop. In our previous recipe this didn't work, because Router 1 didn’t know how to get to 10.10.36.6. However, now that this prefix is in IS-IS, our router can do a recursive lookup on this prefix, and sees that ultimately its next-hop is towards Router 2. We can see this in the output below:

```
user@Router1> show route 203.0.113.0/24

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

203.0.113.0/24     *[BGP/170] 00:01:28, localpref 100, from 3.3.3.3
                      AS path: 64513 I
                    > to 10.10.12.2 via ge-0/0/0.0
```

So long as you’re redistributing this link into your IGP of choice, this setup might be fine. But there’s one definite downside: we’re not running LDP or RSVP between ourselves and this other Internet provider. That means that Router 1 has no entry for 10.10.36.6 in its inet.3 table.

In the output above, notice how the next hop doesn’t involve going down an LSP. Let’s think about the consequences of this. Router 1 looks up this prefix in its routing table, then it:

- Notices the prefix was learned by BGP,
- Sees a protocol next hop of 10.10.36.6,
- Looks in inet.3 for this IP address,
- Finds nothing, so,
- Falls back to the normal inet.0 table,
- Resolves this IP in there instead, and
- Sends the packet out as a normal, unlabeled packet.

There’s a good chance that this isn’t what you want to happen. Remember: for traffic to go down an LSP, by default the BGP protocol next hop *has to be* in inet.3 – and the interface IP address of Router 6, 10.10.36.6, certainly won’t be in there.

That is... unless we put it in there ourselves. The syntax is as easy as pie: just use the install command to associate the next hop IP with the LSP you've made. In our last recipe we made an RSVP label-switched path called TO_ROUTER_3. Let's put this command on this LSP:

```
set protocols mpls label-switched-path TO_ROUTER_3 install 10.10.36.6/32
```

With this one command, we've told Router 1 that if it finds a protocol next hop of 10.10.36.6, it should send it down the LSP we made to Router 3.

Perfect! The next hop 10.10.36.6/32 is now in our inet.3 table, so any prefixes learned by BGP, with a next hop of 10.10.36.6, will go down our LSP:

```
user@Router1> show route 203.0.113.0/24

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

203.0.113.0/24     *[BGP/170] 00:05:29, localpref 100, from 3.3.3.3
                      AS path: 64513 I
                    > to 10.10.12.2 via ge-0/0/0.0, label-switched-path TO_ROUTER_3
```

Now let's take it to the next level: what if you actually want traffic *destined to* 10.10.36.6/32 to go down the LSP as well? At the moment, this isn't happening:

```
user@Router1> show route 10.10.36.6

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

10.10.36.0/24      *[IS-IS/18] 00:05:59, metric 30
                    > to 10.10.12.2 via ge-0/0/0.0
```

This is also simple: all you have to do is add `active` to the end of the install command: `install 10.10.36.6/32 active`:

```
user@Router1> show route 10.10.36.6

inet.0: 15 destinations, 15 routes (15 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

10.10.36.6/32      *[RSVP/7/1] 00:00:02, metric 20
                    > to 10.10.12.2 via ge-0/0/0.0, label-switched-path TO_ROUTER_3
```

There's an important consequence of adding the extra active command, however: it actually takes this prefix *out* of the inet.3 table and puts it *into* the inet.0 table, with a next hop of the LSP. This may or may not be okay, depending on what you're trying to achieve. It won't break BGP – BGP can use both inet.0 and inet.3 to resolve next hops, it just checks inet.3 first. But if an MPLS VPN relies on this protocol next hop being in inet.3, you may find your VPNs no longer work quite as you'd hoped.

Advantage: Gives you the most precision over what goes where.

Disadvantage: This is basically a static route, so it's not very scalable. In addition, using the `active` command might have an impact on MPLS VPNs, as the prefix is in inet.0 instead of inet.3.

Override 2: Advertise an RSVP LSP Into Your IGP

If you like, you can make OSPF or IS-IS believe that your LSP is an actual link, directly connecting the two routers at either end of the path! All you have to do is make LSPs in both directions, then add the LSPs into your IGP, with a metric of your choice. The syntax couldn't be simpler. Once again, let's do it on Router 1:

```
set protocols isis label-switched-path TO_ROUTER_3 level 2 metric 1
```

The equivalent in OSPF would be: `set protocols ospf area 0.0.0.0 label-switched-path TO_ROUTER_3 metric 1`.

You need to add the equivalent command on Router 3 for the LSP in the other direction, too. And when you do, look what happens:

```
user@Router1> show isis adjacency
Interface             System         L State        Hold (secs) SNPA
TO_ROUTER_3           Router3        2  Up                    0
ge-0/0/0.0            Router2        2  Up                    8  50:0:0:4:0:0
ge-0/0/1.0            Router4        2  Up                    7  50:0:0:6:0:1
```

You'll see the LSP being used in SPF calculations:

```
user@Router1> show isis spf brief
 IS-IS level 1 SPF results:
  0 nodes

 IS-IS level 2 SPF results:
Node             Metric     Interface        NH   Via             SNPA
Router5.03       20         ge-0/0/1.0       IPV4 Router4         50:0:0:6:0:1
Router5.00       11         ge-0/0/0.0       LSP  TO_ROUTER_3
Router5.02       11         ge-0/0/0.0       LSP  TO_ROUTER_3
Router3.02       11         ge-0/0/0.0       LSP  TO_ROUTER_3
Router4.00       10         ge-0/0/1.0       IPV4 Router4         50:0:0:6:0:1
Router2.00       10         ge-0/0/0.0       IPV4 Router2         50:0:0:4:0:0
Router3.00       1          ge-0/0/0.0       LSP  TO_ROUTER_3
Router1.00       0
  8 nodes
```

And you can see from the routing table that IS-IS treats the LSP just like an available link. This link has a metric of 1, which gives a total end-to-end metric of 11. The previous non-labelled path of R1-R2-R3 has a metric of 20. Therefore, as you can see below, our router chooses the LSP as the best path:

```
user@Router1> show route 5.5.5.5

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

5.5.5.5/32         *[IS-IS/18] 00:00:50, metric 11
                    > to 10.10.12.2 via ge-0/0/0.0, label-switched-path TO_ROUTER_3
```

Advantage: Gives you even more control over link cost than you had without it.

Disadvantage: Potential for inefficient paths to be preferred over shorter paths, depending on how the LSP is being formed.

Overide 3: Move Everything Into inet.0

R1 has an LSP to R3. As you know, this means that traffic with a BGP next hop of 3.3.3.3 will go down the LSP. But what if you also want all network traffic destined to 3.3.3.3 itself to go down the LSP? Is there a command to do that? Well, in fact, there are a few different ways, starting with this:

```
set protocols mpls traffic-engineering bgp-igp
```

This one command moves – not copies, but moves – the contents of inet.3 into inet.0. LSPs will still be used to resolve BGP next hops – remember, BGP can use both tables, it just prefers inet.3. But now, all your other traffic gets to use the LSP as well!

First, notice that that our inet.3 table is now empty:

```
user@Router1> show route table inet.3

user@Router1>
```

Next, notice that everything is in inet.0 – and because RSVP and LDP have lower route preferences than IS-IS, our router is choosing these routes over our IGP:

```
user@Router1> show route table inet.0 | match 32
1.1.1.1/32         *[Direct/0] 02:19:55
2.2.2.2/32         *[LDP/9] 00:02:03, metric 1
3.3.3.3/32         *[RSVP/7/1] 00:02:03, metric 10
4.4.4.4/32         *[LDP/9] 00:02:03, metric 1
5.5.5.5/32         *[IS-IS/18] 00:02:03, metric 20
10.10.12.1/32      *[Local/0] 02:38:43
10.10.14.1/32      *[Local/0] 02:19:55
```

This is pretty cool stuff. But you might be wondering: if this is so cool, why isn't this the default? Why is it that only BGP traffic gets to use the LSP by default? A good question, with a simple answer: moving everything out of inet.3 breaks your ability to run MPLS VPNs. Don't worry though, there's a way to fix that, too. Keep reading!

Advantage: Only one command to give every non-VRF prefix access to the LSP.

Disavantage: Breaks your ability to do MPLS VPNs.

Override 4: Copy Everything from inet.3 to inet.0

This command does the same as the one above, but instead of *moving* the contents of inet.3 to inet.0, it *copies* it:

```
set protocols mpls traffic-engineering bgp-igp-both-ribs
```

This means that your normal prefixes can use the LSP *and* you can still run MPLS VPNs!

If you do a lookup in Router 1's routing tables for Router 2's loopback, you'll see an LDP entry in both inet.0 and inet.3:

```
user@Router1> show route 2.2.2.2

inet.0: 14 destinations, 17 routes (14 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

2.2.2.2/32         *[LDP/9] 00:00:07, metric 1
                    > to 10.10.12.2 via ge-0/0/0.0, Push 0
                    [IS-IS/18] 00:02:53, metric 10
                    > to 10.10.12.2 via ge-0/0/0.0

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

2.2.2.2/32         *[LDP/9] 00:00:07, metric 1
                    > to 10.10.12.2 via ge-0/0/0.0, Push 0
```

Again, you might be wondering: why isn't *this* the default? This seems like the best of both worlds.

Well, as with all of these solutions, there's a downside: when you activate this command and you make an LSP the preferred route in your inet.0 table, it can seriously break your IGP. In short, your IGP routes are no longer the *active* routes because now our LSP is the best route instead – and if our *normal* routes are no longer the best, this might stop them from being advertised around the rest of the network. Luckily, there's a fix for that, too.

Advantage: Gives you loads of functionality, including MPLS VPNs.

Disadvantage: Can break your routing policies.

Override 5: Use the LSP Just for Forwarding and Use Your IGP Just for Advertising Routes

What if there was a command that copied LSPs into inet.0 for forwarding but didn't use these LSPs when it came to making routing protocol decisions – and that this command also kept the IP in inet.3 as well for VPN use?

Consider today a birthday and Christmas combined, because there is a final command in this recipe:

```
set protocols mpls traffic-engineering mpls-forwarding
```

This command allows your router to use LSPs for forwarding traffic but ignores them when it comes to the way the router advertises prefixes into your IGP.

Take a look at this output, and you'll notice a few new symbols in the legend, like `@` and `#`. This shows us that traffic will be forwarded down the LSP, but IS-IS still uses its own *best* path for a consistent IS-IS topology:

```
user@Router1> show route 5.5.5.5

inet.0: 14 destinations, 19 routes (14 active, 1 holddown, 0 hidden)
@ = Routing Use Only, # = Forwarding Use Only
+ = Active Route, - = Last Active, * = Both

5.5.5.5/32         @[IS-IS/18] 00:00:01, metric 20
                    > to 10.10.14.4 via ge-0/0/1.0
                   #[LDP/9] 00:00:01, metric 1
                    > to 10.10.14.4 via ge-0/0/1.0, Push 299840

inet.3: 4 destinations, 5 routes (4 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

5.5.5.5/32         *[LDP/9] 00:00:01, metric 1
                    > to 10.10.14.4 via ge-0/0/1.0, Push 299840
```

So, yet again, you might be thinking: *Okay, this really does sound perfect. Why isn't THIS* the *default*? Well, it sure is good, but the problem is that it's an all-or-nothing approach. This command copies everything into inet.0, which doesn't give you much control over precisely what goes in. If you do want everything to go in, great! But in an ISP-sized network, you probably don't. When you think of it like that, it's a little easier to see why this isn't the default behavior – no matter how attractive it might seem.

Advantage: Lets you copy inet.3 to inet.0 without it affecting your IGP's routing decisions.

Disadvantage: Doesn't give you much granularity in choosing exactly which traffic should take your LSP.

Discussion

These are just five of the ways you can manipulate what traffic goes down an LSP. If you want even more control, there are even more commands available to you, including some complicated but brilliant ways of using RIB groups.

Recipe 5: Setup, Best Practices, and Pitfalls of MC-LAG on the QFX-Series

By Christian Scholz

- vQFX JunOS Version Used: 18.4R1.9
- Juniper Platforms General Applicability: vQFX, QFX, EX

If you are one of the people that run away in fear when you hear *MC-LAG,* don't worry anymore – this recipe is just for you. MC-LAG can be a real pain to set up and tweak, but once you get it running, it can last forever without the need to touch it again.

IMPORTANT This lab was tested inside a virtual environment on the vQFX. Some settings (especially the timers) have been adjusted to work best inside that virtual environment. Depending on your lab setup, you may need to carefully adjust the timers. It's always recommended, and best practice, to test this inside a lab environment to verify your design.

Problem

Let's assume you have a core or distribution layer with multiple, "single" devices running on Junos – maybe as part of the IP-fabric that you designed.

You want to provide redundant link paths towards the core / distribution from the access layer point of view for Layer 2 traffic. Configuring a simple LAG will not work, because the links will connect towards two peers at the core / distribution layer and the LAG will never come up because the core / distribution peers don't recognize each other to balance across the LAG. What to do now?

You already guessed it: MC-LAG. MC-LAG technology enables you to establish a logical LAG connection towards a redundant pair of independent peers. And the best part is the peers may even run different Junos versions – so during your upgrades it is possible to update each core / distribution device one after another without losing connectivity. How cool is that?

The access device isn't even aware of MC-LAG running inside your core – from the access layer's point of view you just configure a simple LAG towards the core / distribution. As shown in Figure 5.1, the two MC-LAG peers will act as a single switch towards the access-switch and therefore the LAG will work fine.

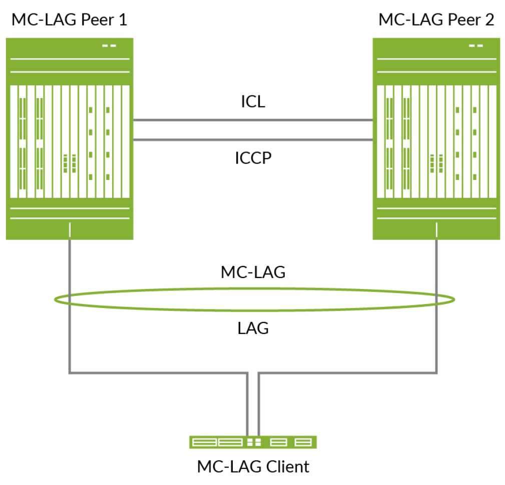


Figure 5.1 *MC-LAG Basic View*

Troubleshooting and tweaking MC-LAG can be painful if you don't know where to look, but don't worry – this recipe should help.

Solution

Best practices for MC-LAG on the QFX and EX Series cover the following:

- What pitfalls to look for (after including an example for you to use).
- What timers to use.
- Information regarding ICCP and ICL-PL.
- How to use VRRP with MC-LAG.

For this recipe let's use the topology shown in Figure 5.2, assuming that MC-LAG peer1 and peer2 are vQFX Series running on 18.4R1.9 and the MC-LAG client is serving as a "generic switch" that knows nothing about the MC-LAG config and simply has one LAG configured towards both peers. This client can be an EX, QFX, MX, or SRX-Series, or any other vendor switch that supports LAGs.

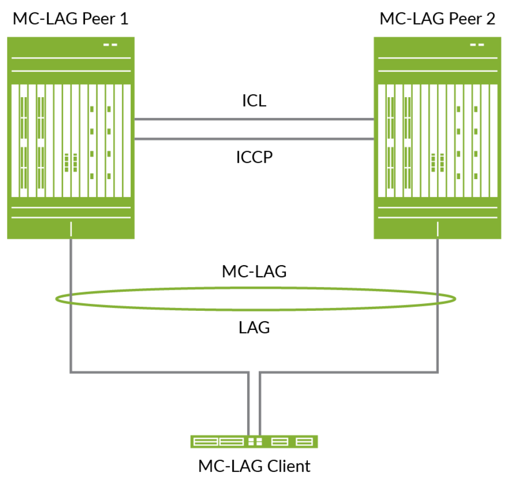


Figure 5. 2 Recipe 5 Lab Topology

Example and Pitfalls

When configuring MC-LAG always remember that the settings in Table 5.1 must match and be unique in members of the MC-LAG.

Table 5.1 MC-LAG Settings

Setting	Info
LACP System-ID	Must match on both Peers
LACP Admin-Key	Must match on both Peers
MCAE-ID	Must match on both Peers
MCAE-Mode	Must match on both Peers
VLAN’s (ICL-PL and Access)	Must match on both Peers
MCAE-Chassis ID	Must be unique for each Peer
MCAE Status Control	Must be unique for each Peer
ICCP-IP (Local)	Must be unique for each Peer
ICCP-IP (Peer)	Must be unique for each Peer
MC-LAG Protection	Must be unique for each Peer

Ninety-nine percent of MC-LAG troubleshooting is about the configuration of your MC-LAG – it can get so big that you simply forget to include a single statement, and that causes MC-LAG to flap or completely refuse to work. If you only work with MC-LAG from time to time, it’s best to have an example snippet to work with; so this recipe creates a simple MC-LAG topology to follow in the lab. In reality you can have multiples of these running on the same peers – but if you look at the simple config you can already tell how many lines you will get.

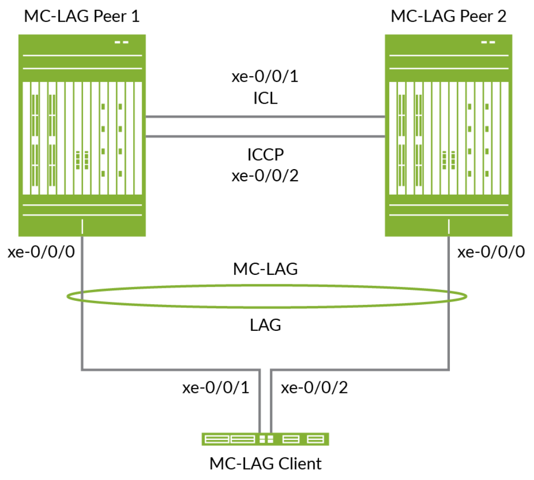


Figure 5.3 Recipe 5 Lab Topology with Interface-mapping

Let's start with the configuration:

```
[edit]
root@Peer-1# set chassis aggregated-devices ethernet device-count 2
```

This is by far the most common mistake – you create another MC-LAG ae, configure all settings, and still wonder why your interface is not showing up:

```
[edit]
root@Peer-1# set interfaces xe-0/0/1 ether-options 802.3ad ae0
root@Peer-1# set interfaces xe-0/0/2 ether-options 802.3ad ae0
```

Always make sure to use redundant ICL-PL links with at least two members:

```
[edit]
root@Peer-1# set interfaces xe-0/0/0 ether-options 802.3ad ae1
```

Your link towards the access-device will be one member-link per MC-LAG device / core:

```
[edit]
root@Peer-1# set interfaces ae0 unit 0 family ethernet-switching interface-mode trunk
root@Peer-1# set interfaces ae0 unit 0 family ethernet-switching vlan members vl-iclpl
root@Peer-1# set interfaces ae0 unit 0 family ethernet-switching vlan members vl-access
```

On the ICL-PL you need to add your control-VLAN (for your ICCP protocol to exchange all necessary information) and also all the VLANs that your MC-LAG client uses – this is useful if you have traffic that needs to cross your peers in order to get to a device that's only connected to one peer in an active-active setup. This depends on your design:

```
[edit]
root@Peer-1# set interfaces ae1 aggregated-ether-options lacp active
root@Peer-1# set interfaces ae1 aggregated-ether-options lacp system-id 00:01:01:01:01:01
root@Peer-1# set interfaces ae1 aggregated-ether-options lacp admin-key 1
root@Peer-1# set interfaces ae1 aggregated-ether-options mc-ae mc-ae-id 1
root@Peer-1# set interfaces ae1 aggregated-ether-options mc-ae chassis-id 1
root@Peer-1# set interfaces ae1 aggregated-ether-options mc-ae mode active-active
root@Peer-1# set interfaces ae1 aggregated-ether-options mc-ae status-control standby
root@Peer-1# set interfaces ae1 aggregated-ether-options mc-ae init-delay-time 240
root@Peer-1# set interfaces ae1 aggregated-ether-options mc-ae redundancy-group 1
root@Peer-1# set interfaces ae1 unit 0 family ethernet-switching interface-mode trunk
root@Peer-1# set interfaces ae1 unit 0 family ethernet-switching vlan members vl-access
```

Set the necessary values for your environment (remember that some values mentioned earlier must match, while others must be different) and check if you need LACP. In case your MC-LAG client (a server for example) can't understand LACP, but you still want to use it, you can "trick" MC-LAG by adding the `force-up` option for LACP. This way LACP will come up even if your server is not aware of LACP:

```
[edit]
root@Peer-1# set vlans vl-iclpl vlan-id 1337
root@Peer-1# set vlans vl-access vlan-id 1007
```

Define your VLANs on the MC-LAG peers and the MC-LAG client. The ICL-PL VLAN has to be created only on the MC-LAG-peers – the client only has the local significant VLANs created and used on this uplink:

```
[edit]
root@Peer-1# set multi-chassis multi-chassis-protection 3.3.3.2 interface ae0
root@Peer-1# set protocols iccp local-ip-addr 3.3.3.1
root@Peer-1# set protocols iccp peer 3.3.3.2 redundancy-group-id-list 1
root@Peer-1# set protocols iccp peer 3.3.3.2 liveness-detection minimum-receive-interval 900
root@Peer-1# set protocols iccp peer 3.3.3.2 liveness-detection transmit-interval minimum-
interval 900
root@Peer-1# set switch-options service-id 1
```

Timers

Depending on your design and device types being part of MC-LAG, you should avoid timers for liveness detection smaller than 500ms. This also depends on where your MC-LAG is used. For larger environments with multiple MC-LAGs connected to your MC-LAG-peers, 500ms has proven to be a good value – but again this strongly depends on your environment and your devices being used. Start with larger timers and adjust them slowly towards your goal.

ICCP / ICL-PL

MAC address synchronization enables MC-LAG peers to forward Layer 3 packets arriving on multichassis aggregated Ethernet interfaces with either their own IRB or RVI MAC address, or their peer's IRB or RVI MAC address. If MAC address synchronization is not enabled, the IRB or RVI MAC address is installed on the MC-LAG peer as if it were learned on the ICL. You can do this by enabling `mcae-mac-synchronize` on the ICL-PL VLAN (see ae0 on the example earlier).

Since this ICL-PL is your "heartbeat," you want to make sure to use a redundant ae with at least two members. It's also recommended to connect both peers directly with each other and not run the ICL-PL over another pair of devices – side-effects may occur.

VRRP

For the QFX Series, Juniper recommends that you use MAC address synchronization for the downstream clients. For the upstream routers, they recommend that you use VRRP over IRB or the RVI method. Why, you ask? That's because on the QFX Series, routing protocols are not supported on the downstream clients.

If you are using the VRRP over IRB method to enable Layer 3 functionality, you must configure static ARP entries for the IRB interface of the remote MC-LAG peer to allow routing protocols to run over the IRB interfaces. This step is required so you can issue the ping command to reach both the physical IP addresses and virtual IP addresses of the MC-LAG peers and allow VRRP to work. If you forget about this step, the "funniest" side effects may occur, or VRRP will refuse to run completely!

Discussion

There's so much more to say about MC-LAG, but this short recipe has given you some guidelines and an overview of common pitfalls. MC-LAG can be a good design choice if you want the flexibility to upgrade each peer separately while not losing your connection. There are also technologies like EVPN – but if you don't want to scale out, MC-LAG can be the perfect fit for you. Use the included sample configuration from the lab and adjust it to your needs.

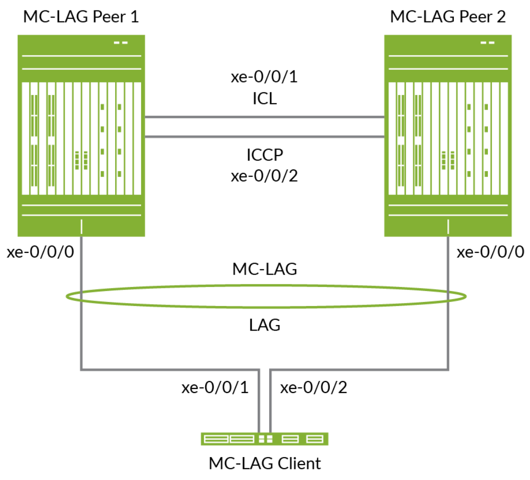


Figure 5.4 Recipe 5 Topology and Lab Configuration

Recipe 6: Connecting an SRX Cluster to a VRRP Router

By Michel Tepper

- Junos OS Used: 18.4R1
- Juniper Platforms General Applicability: branch and mid-range SRX

A midsize software company started offering their solution as a service. They rented half a rack in a data center, put in some storage, a hypervisor, a switch, and an SRX. The setup wasn't too difficult, but it also wasn't very redundant.

So, the company decided to invest in redundancy. The SRX was made into a cluster, and the Internet uplink was made redundant. The network was made redundant. Single interfaces where replaced by reth interfaces. On the untrust side it looked like this:

```
root@fw1-n0# show chassis cluster
reth-count 2;
redundancy-group 0 {
    node 0 priority 200;
    node 1 priority 100;
}
redundancy-group 1 {
    node 0 priority 200;
    node 1 priority 100;
    preempt {
        delay 5;
    }
    interface-monitor {
        ge-0/0/7 weight 255;
        ge-0/0/6 weight 255;
    }
}

root@fw1-n0# show chassis cluster
reth-count 2;
redundancy-group 0 {
    node 0 priority 200;
    node 1 priority 100;
}
redundancy-group 1 {
    node 0 priority 200;
    node 1 priority 100;
    preempt {
        delay 5;
    }
    interface-monitor {
        ge-0/0/7 weight 255;
        ge-0/0/6 weight 255;
    }
}

root@fw1-n0# show interfaces fab0
fabric-options {
    member-interfaces {
        ge-0/0/2;
    }
}
root@fw1-n0# show interfaces fab1
fabric-options {
    member-interfaces {
        ge-5/0/2;
    }
}

primary:node0}[edit]
root@fw1-n0# show interfaces reth1
redundant-ether-options {
    redundancy-group 1;
}
unit 0 {
    family inet {
        address 100.31.5.226/24;
    }
}

{primary:node0}[edit]
root@fw1-n0# show interfaces ge-0/0/7
gigether-options {
    redundant-parent reth1;
}

{primary:node0}[edit]
root@fw1-n0# show interfaces ge-5/0/7
gigether-options {
    redundant-parent reth1;
}

root@fw1-n0# show security zones security-zone untrust
screen untrust-screen;
interfaces {
    reth1.0 {
        host-inbound-traffic {
            system-services {
                dhcp;
                tftp;
                ssh;
                http;
                https;
            }
        }
    }
}
```

The fab and control links were established, the company connected the redundant Internet uplink, and then they were surprised by a huge packet loss! Huge here means at least a 75% loss. The company then disconnected one of the links and everything was fine again. Since the company's core business is software development, not networking, they hired a consultant to explain what was going on. After a short evaluation the consultant asked: "Is this redundant Internet uplink based upon VRRP?" After looking through the offering from the ISP the answer was "Yes."

Problem

It then became clear that the packet loss was caused by duplicate IP addresses in the network. The two routers from the ISP were connected over reth interfaces. On a reth interface, per the standard, one link is active, and one passive. This means the multicast traffic VRRP uses to monitor the peer status can't reach each other. Both routers "think" they are the master and advertise the shared IP address. It's this duplicate address on the network that results in the packet loss as shown in Figure 6.1.

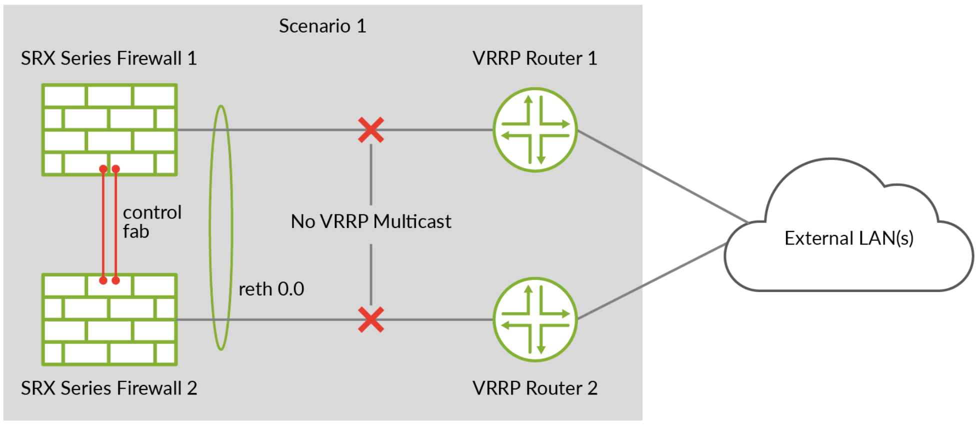


Figure 6.1 Duplicate IP Addresses in the Network

Solution

Solution 1: A simple solution is connecting both nodes from the SRX cluster to a switch and placing them in one VLAN, together with the two connections to the ISP. But since redundancy is the primary goal, from this perspective using one switch to connect both uplinks isn't a good idea. To avoid introducing a single point of failure, two switches need to be used, as in Figure 6.2.

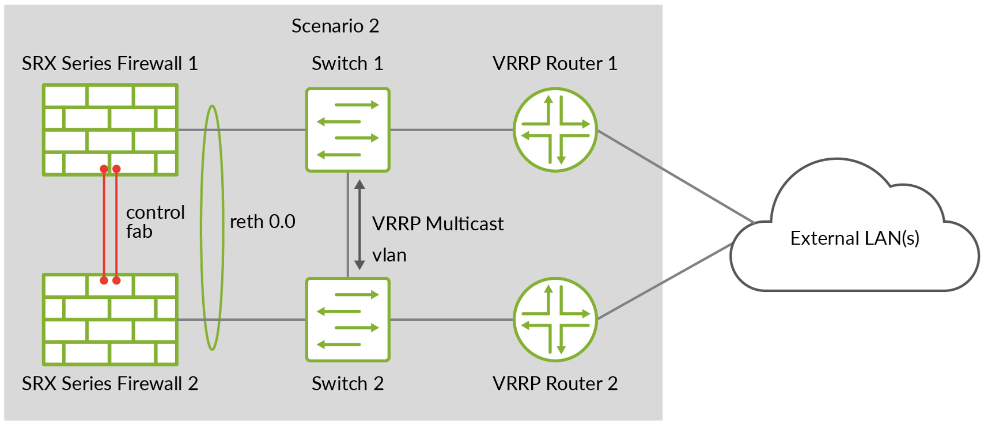


Figure 6.2 Solution 1: Using Two Switches

This ensures that both members of the SRX cluster can connect to both VRRP routes, and the routers can monitor each other's status with the multicast traffic. It's a good solution, but it comes at the cost of two extra switches. The switches need to be acquired, they occupy rack space, they consume power, etc. Surely there must be another way.

Solution 2: The second solution is to solve the problem without the external switch(es). Instead, make use of the built-in switching capabilities of the SRX as decribed in Knowledge Base article KB21422. It takes a little configuration, but it isn't hard to understand.

NOTE The first step in this configuration is to put the SRX in switching mode. On older versions of SRX (pre-Junos 15.1X49-D40), as well as on new versions (15.1X49-100 and higher), this is the default behavior. Because of the dependency on version numbers, it's a good idea to just configure:

```
set protocols l2-learning global-mode switching
```

Having done this, you need to add an extra link between the two members of the cluster. You already have the control link and the fab link. The control link is used for control traffic, the fab link for Layer 3 traffic, which might arrive on one node and is routed using an outgoing interface on the other node. The third link to create is the swfab (switching fabric) link. This link binds the two switching domains from the SRXs together. It's kind of like the VC-link in a virtual chassis of switches. From the configuration it looks like the fab-link and multiple links may be used and they will form a LAG when configured.

The fab and swfab are configured this way:

```
root@fw1-n0# show interfaces swfab0
fabric-options {
    member-interfaces {
        ge-0/0/3;
    }
}

root@fw1-n0# show interfaces swfab1
fabric-options {
    member-interfaces {
        ge-5/0/3;
    }
}
```

Configuring this link enables you to join the two switching domains. Let's check with the `show chassis cluster ethernet-switching status` command:

```
root@fw1-n0> show chassis cluster ethernet-switching status
Monitor Failure codes:
    CS  Cold Sync monitoring        FL  Fabric Connection monitoring
    GR  GRES monitoring             HW  Hardware monitoring
    IF  Interface monitoring        IP  IP monitoring
    LB  Loopback monitoring         MB  Mbuf monitoring
    NH  Nexthop monitoring          NP  NPC monitoring
    SP  SPU monitoring              SM  Schedule monitoring
    CF  Config Sync monitoring      RE  Relinquish monitoring

Cluster ID: 1
Node   Priority Status               Preempt Manual   Monitor-failures

Redundancy group: 0 , Failover count: 1
node0  200      primary              no      no       None
node1  100      secondary            no      no       None

Redundancy group: 1 , Failover count: 3
node0  200      primary              yes     no       None
node1  100      secondary            yes     no       None

Ethernet switching status:
    Probe state is UP. Both nodes are in single ethernet switching domain(s).
```

The last line of the output clearly shows the swfab link is working. If it was *not* working, it would show:

```
Ethernet switching status:
    Probe state is DOWN. Both nodes are in separate ethernet switching domain(s).
```

Now that there's a working link and one common switching domain, the next step is to configure a VLAN for the untrusted side and put the VRRP routers in this VLAN. This means you're deleting the reth interface, reth1, and removing it from the untrust security zone. The member links are placed into the VLAN, and the configuration for the untrusted side now looks like the following:

```
root@fw1-n0# show vlans untrust-vlan
vlan-id 999;
l3-interface irb.999;
```

```
{primary:node0}[edit]
root@fw1-n0# show interfaces irb
unit 999 {
    family inet {
        address 100.31.5.226/24;
    }
}

{primary:node0}[edit]
root@fw1-n0# show interfaces ge-0/0/7
unit 0 {
    family ethernet-switching {
        vlan {
            members untrust-vlan;
        }
    }
}

{primary:node0}[edit]
root@fw1-n0# show interfaces ge-5/0/7
unit 0 {
    family ethernet-switching {
        vlan {
            members untrust-vlan;
        }
    }
}
```

After committing this configuration, you can check things by looking at the VLAN:

```
{primary:node0}[edit]
root@fw1-n0# run show vlans untrust-vlan

Routing instance        VLAN name             Tag          Interfaces
default-switch          untrust-vlan          999
                                                           ge-0/0/7.0*
                                                           ge-5/0/7.0*
```

The two VRRP routers connect to two access ports on the same VLAN. The multicast packets they are transmitting can now reach the partner device, and the SRX can reach both VRRP routers from its L3 VLAN interface (irb.999). This is the interface you need to place in the security zone:

```
root@fw1-n0# show security zones security-zone untrust
screen untrust-screen;
interfaces {
    irb.999 {
        host-inbound-traffic {
            system-services {
                dhcp;
                tftp;
                ssh;
                http;
                https;
            }
        }
    }
}
```

The configured solution now looks like this:

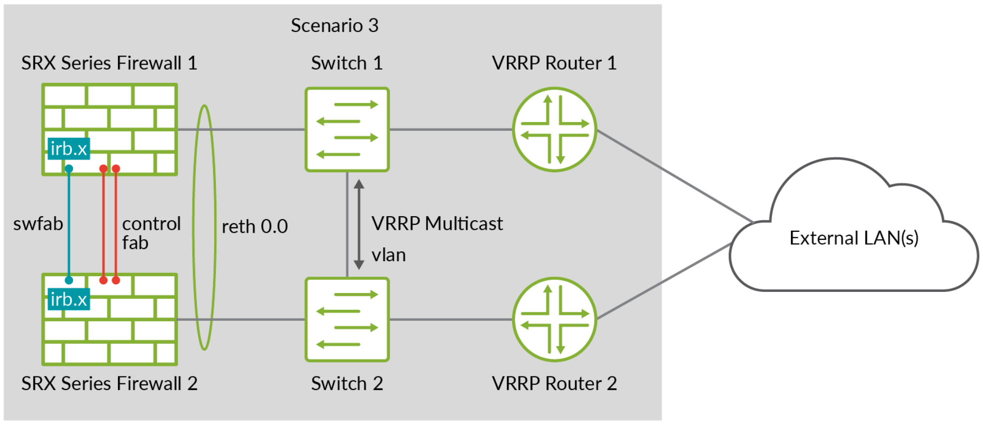


Figure 6.3 Solution 2: Two VRRP Routers XConnect to Two Access Ports on the Same VLAN

Discussion

Solution 1 made use of two external switches and a simple cluster configuration. Solution 2, on the other hand, solves the problem posed in this recipe without extra devices. But there is a downside. In the case of losing a connection to one of the routers, the reaction time will be slower. This is nothing to worry about; the failover of the VRRP will also be around three seconds. For most applications this is far within reasonable limits. The second downside might be the extra link(s) between the devices. On a SRX300/320 with six copper ports this leaves two ports to use freely. The best approach in this case might be to use a tagged interface on a reth interface with two members from each node connecting to a trunk LAG pair on a switch.

On the 340/345 and above, the one or two extra ports for the swfab shouldn't matter.

Recipe 7: Consolidation of Two PEs: BGP Pre- and Post-Check With PyEZ

By Nupur Kanoi

- Python Version Used: 2.7
- Junos OS Used: 16.1
- Juniper Platforms General Applicability: MX Series, vMX

Juniper has developed a microframework for Python called PyEZ, which enables a network engineer to gather required details from multiple devices without logging in manually to those devices. PyEZ also reduces the complexity of getting unnecessary information by allowing you to pull only the required details with the help of RPC calls. This recipe makes use of RPC calls to implement the solution to the problem posed below.

Problem

Confirming a successful integration of two provider edge routers (PEs) can be a daunting job in terms of BGP pre- and post-checks. A situation in which there are redundant BGP peers in two PEs and reusing same private IP address in multiple customer VRFs makes it tricky. This recipe simplifies the task by automating the pre- and post-check for the integration of the two PEs.

Solution

Getting ready:

1. Install PyEZ with the help of this link:

https://www.juniper.net/documentation/en_US/junos-pyez/topics/task/installation/junos-pyez-server-installing.html

MORE? Also see the *Day One: Junos PyEZ Cookbook* at: https://www.juniper.net/us/en/training/jnbooks/day-one/automation-series/junos-pyez-cookbook/.

2. Install the modules below in order for the upcoming script to work:

- Install the “lxml” module: This is the library to process XML in Python. With the help of this module you will convert the XML element tree to dictionary format, which is a more convenient way of dealing with information than an unordered name-value pair in XML format.
- Install the “jxmlease” module: This is the library used to parse dictionary data.

3. Determine the RPCs to be used to gather the required information. Juniper Networks has a great tool that helps you to check the RPC that needs to be used: https://apps.juniper.net/xmlapi/operTags.jsp . Alternatively, you can also get the same information from the device, as shown below. The bolded portions show the RPCs that you will be using in this recipe:

```
lab@MX480> show bgp summary | display xml rpc
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/16.1R4/junos">
    <rpc>
        <get-bgp-summary-information>
        </get-bgp-summary-information>
    </rpc>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>

lab@MX480> show bgp neighbor | display xml rpc
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/16.1R4/junos">
    <rpc>
        <get-bgp-neighbor-information>
        </get-bgp-neighbor-information>
    </rpc>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>
```

NOTE This recipe uses the default SSH port 22 for NETCONF to device. SSH by default allows NETCONF over it, hence there is no need to specifically enable NETCONF on the device if you’re using port 22 for NETCONF session. The only command required to access the device is `set system services ssh`.

Pre-Check

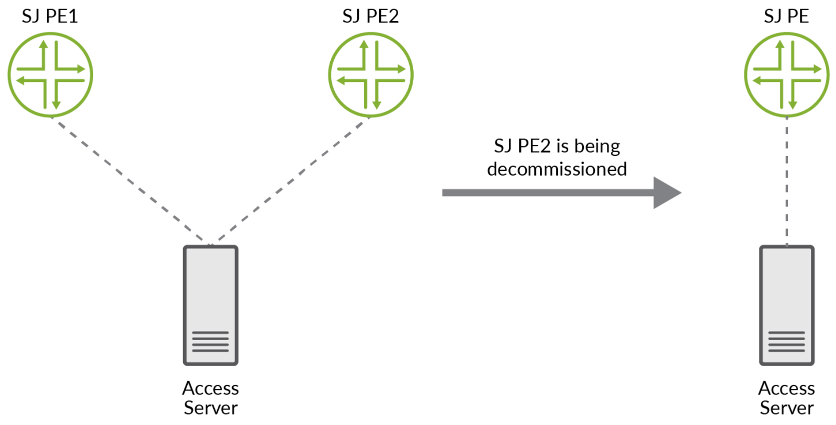


Figure 7.1 Recipe 7 Topology Depicting the Before and After Scenarios

Here is the pre-check script:

```
#!usr/bin/python

from jnpr.junos import Device                                          #1
from lxml import etree
import jxmlease


r1 = raw_input("PE1: ")                                                #2
r2 = raw_input("PE2: ")
HOST = [r1,r2]                                                           #3
f1 = ""
f2 = ""
y = None
j = 0      # to generate file-name variable
k1 = []
k2 = []

#For Final consolidation
N = []



for item in HOST:
          print("\n")

          with Device(host=item, user='lab', password='lab', port=22) as dev:
            print("Logged in " + dev.facts['hostname'] + " | Model = " + str(dev.facts['model'])) #4

            if j == 0:                                                     #5
              f1 = str(dev.facts['hostname'] + "_OLD")
```

```
  y = open(f1,"w+")
if j == 1:
  f2 = str(dev.facts[‘hostname’] + "_OLD")
  y = open(f2,"w+")


r = dev.rpc.get_bgp_summary_information(normalize=True, dev_timeout=55)      #6
Peer = r.xpath("//peer-count")[0].text
Down = r.xpath("//down-peer-count")[0].text
Estab = int(Peer) - int(Down)                      # determine Establish peer count
Est = []
Dwn = []


print("Total Estab= {} Down Peer = {}".format(Estab,Down))
rsp = dev.rpc.get_bgp_neighbor_information(normalize=True, dev_timeout=55)      #7
rpc_xml = etree.tostring(rsp, pretty_print=True, encoding=’unicode’)
print("Hang on.. Parsing data, might take few minutes")

xmlparser = jxmlease.Parser()
result = jxmlease.parse(rpc_xml)

uniq_list = []                                                                #8
for neighbor in result.find_nodes_with_tag(‘bgp-peer’):               #9

  Neighbor = neighbor[‘peer-address’].split(‘+’)[0]
  State = neighbor[‘peer-state’]

  for item in neighbor.find_nodes_with_tag(‘bgp-rib’):

    Table = item[‘name’]
    if Table == ‘bgp.l3vpn.0’:
      Est.append("{} {} {}".format(Neighbor,Table,State))
      uniq_list.append(Neighbor)

  for item in neighbor.find_nodes_with_tag(‘bgp-rib’):
    Table = item[‘name’]
    if Neighbor not in uniq_list :
      Est.append("{} {} {}".format(Neighbor,Table,State))


  if State != "Established":                                                  #10
    Dwn.append("{} down".format(Neighbor))


if j == 0:                                                                    #11
  print("Data stored in file {}".format(f1))
  for i in Est:
    k1.append(i)
  for i in Dwn:
    k1.append(i)
  for i in k1:
    y.write(i + "\n")
  y.close()
if j == 1:
  print("Data stored in file {}".format(f2))
  for i in Est:
    k2.append(i)
  for i in Dwn:
```

```
                  k2.append(i)
                for i in k2:
                  y.write(i + "\n")
                y.close()
              j += 1


print("Processing consolidated BGP output file")

U = ("Consolidated_BGP_file")
d = open(U, "w+")


di = list(set(k1) - set(k2))                                                        #12

for item in k2:                                                                     #13
          N.append(item)
for item in di:
          N.append(item)

for item in N:                                                                      #14
          d.write("{}\n".format(item))


print("Consolidated BGP output stored in file '{}'".format(U))
```

Okay, now let's see what's happening in the pre-check script using the output numbering:

#1. Import standard PyEZ class `Device`, `etree` from lxml and `jxmlease`.

#2. Take user inputs for PE1 and PE2 (IP addresses).

#3. Define the `variables`, which will be further used in the script:

- "HOST" list to store the user input mentioned in step 2.
- "f1" and "f2" are the empty global variables defined to auto-generate the file name based on the hostname of PEs.
- "y" variable, which will be used later to store the current file name.
- "j" variable to incremental value, which helps to generate the current file name and store data in the respective file.
- "k1" and "k2" are empty lists, which later store the BGP output of PE1 and PE2, respectively, in list format to avoid file operation multiple times.
- "N" is an empty list that will later have a consolidated BGP output in list format from which the data will be copied to the final file.

#4. After logging in with a NETCONF connection (port 22 is specifically mentioned to avoid using default NETCONF-over-ssh port 830) to device `dev.facts` helps to retrieve basic information of the logged-in device. It gives output in Python dictionary format, from which we are printing the hostname and model of the device.

#5. Again with the help of device attribute `'dev.facts'` we will be generating a file name.

#6. We are using `"show bgp summary"` XML RPC to get the total number of established and down peer counts. The RPC is `'get-bgp-summary-information'` but the catch here is we need to replace '-' with '_' while using these RPCs in PyEZ, hence the RPC becomes `'get_bgp_summary_information'`

#7. We are using `'get_bgp_neighbor_information'` RPC to get more granular details like peer address with an associated table and its state. Further we are converting the `xml etree` data to dictionary format, which will be parsed with the help of `jxmlease parser`.

#8. Our motive is to remove the duplicate MP-BGP peer found in both the PEs, while retaining all peer addresses used in different customer VRFs, even if they are the same. Therefore, we first gather details for neighbors having the bgp.l3vpn.0 table (which will be MP-BGP peer) and simultaneously we will add that neighbor IP address to `uniq_list` so that the MP-BGP peers will be excluded while looking for customer-specific details. This is done because MP-BGP peers will also have the customer routing table associated with it as a secondary routing table, and that's currently unnecessary information for us.

#9. To achieve the logic explained in point 8 we are going to look for tag `'bgp-peer'` in the information collected with the help of `'get_bgp_neighbor_information'` RPC. Each `'bgp-peer'` tag will have BGP peer details.

#10. When the neighbor state is down, by default there is no table associated with it, even when it's part of a customer VRF, therefore we need to store the down neighbor's detail separately.

#11. Here we are simply appending the collected BGP information in respective files.

#12. `'k1'` and `'k2'` entails the PE1 and PE2 BGP information. Though they are in list format we are using the set feature to get the difference between the two list contents. The result `'di'` will be non-duplicates between PE1 and PE2 present in PE1. The decision of duplication depends on IP address, table as well as state. All 3 values must me same in case it to be considered as duplicates. The functionality is depicted below for better understanding:

```
>>> a = [1,3,4]
>>> b = [1,3,5]
>>> print(set(a)-set(b))
set([4])
```

#13. The final consolidated file should have PE2 output and the non- duplicates that were present in PE1, therefore first appending `'k2'` list contents and then `'di'` list contents to a list `'N'`, with `'N'` as the final expected output.

#14. The final action is to copy the contents in `'N'` to the file named `'Consolidated_BGP_file'` to make it more readable to humans.

Execution of the pre-checks:

```
[Nupur@localhost ~]$ ./pre-check_BGP_v1_lab.py
PE1: 192.168.0.123
PE2: 192.168.0.45

Logged in SJ-MX480 | Model = MX480
Total Estab= 305 Down Peer = 5
Hang on.. Parsing data, might take few minutes
Data stored in file SJ-MX480_OLD

Logged in SJ-PE2 | Model = MX240
Total Estab= 128 Down Peer = 2
Hang on.. Parsing data, might take few minutes
Data stored in file SJ-PE2_OLD
Processing consolidated BGP
Consolidated BGP output stored in file 'Consolidated_BGP_file'
```

In a nutshell, the pre-check script will generate three files: one for PE1, one for PE2, and one for the consolidated BGP output called "Consolidated_BGP_file". The consolidated output file name will be required as a user input in the post-check script.

Post-Check

Figure 7.2 illustrated the final topology.

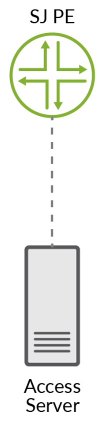


Figure 7.2 Recipe 7 Final Topology After Consolidation

And here's the post-check script:

```
#!usr/bin/python

from jnpr.junos import Device                                      #1
from lxml import etree
import jxmlease
```

```
HOST = raw_input("PE: ")

Pre_check = raw_input("Please specify pre-check file-name: ")

C = []
k1 = []
name = ‘’

with Device(host=HOST, user=’lab’, password=’lab’, port=22) as dev:
          print("\n")
          print("Logged in " + dev.facts[‘hostname’] + " | Model = " + str(dev.facts[‘model’]))

          name = str(dev.facts[‘hostname’])

          y = open(name,"w+")                   # Create file

          r = dev.rpc.get_bgp_summary_information(normalize=True, dev_timeout=55)
          Peer = r.xpath("//peer-count")[0].text
          Down = r.xpath("//down-peer-count")[0].text
          Estab = int(Peer) - int(Down)
          print("Total Estab = {} Down Peer = {}".format(Estab,Down))

          rsp = dev.rpc.get_bgp_neighbor_information(normalize=True, dev_timeout=55)
          rpc_xml = etree.tostring(rsp, pretty_
print=True, encoding=’unicode’) # converting to dictionary format

          print("Hang on.. Parsing data, might take few minutes")

          xmlparser = jxmlease.Parser()
          result = jxmlease.parse(rpc_xml)  # to parse through dictionary name-values

          Est = []
          Dwn = []
          uniq_list =[]

          for neighbor in result.find_nodes_with_tag(‘bgp-peer’):
            Neighbor = neighbor[‘peer-address’].split(‘+’)[0]
            State = neighbor[‘peer-state’]

            for item in neighbor.find_nodes_with_tag(‘bgp-rib’):
              Table = item[‘name’]
              if Table == ‘bgp.l3vpn.0’:
                Est.append("{} {} {}".format(Neighbor,Table,State))
                uniq_list.append(Neighbor)

            for item in neighbor.find_nodes_with_tag(‘bgp-rib’):
              Table = item[‘name’]
              if Neighbor not in uniq_list:
                Est.append("{} {} {}".format(Neighbor,Table,State))

            if State != "Established":
              Dwn.append("{} down".format(Neighbor))

          for i in Est:
            k1.append(i)
          for i in Dwn:
            k1.append(i)
          for i in k1:
            y.write(i + "\n")
          y.close()
          print("Data stored in file {}".format(name))


print("Comparing " +  str(Pre_check) + " and " + name)                          #2

U = str("Missing_in_new_PE")
d = open(U,"w+")

with open(Pre_check) as a:
          for y in a:
            C.append(y.strip())

di = list(set(C) - set(k1))                                                       #3

if di:                                                                                    #4
           d.write("--------------------\nState expected in Post-checks\n--------------------
\n" )
           for item in di:
             d.write("{}\n".format(item))

print("Difference output is stored in file '{}'".format(U))
```

Let's see what is happening in the post-check script:

#1. The first step is pretty much the same as in the first section of the pre-check script, except for the user inputs – one is the the new PE (log ins once in new PE) and the other is the consolidated file name generated with the help of the pre-check script.

#2. Using the first step, a new file with the same name as the hostname is generated, which will be used to compare with the pre-check file called `'Consolidated_BGP_file'`. The content of both the new file generated from step 1 and `'Consolidated_BGP_file'` is expected to be same but in the case it's missing anything, then "Step 3" will throw out the difference.

#3. Using the same technique used in step 12 in the pre-check script, you get contents that are missing from the new PE file.

#4. Finally, the script writes the results from step 3 to a final file called `'Missing_in_new_PE'`.

The execution of the post-check script:

```
[Nupur@localhost ~]$ ./post-check_BGP_v1_lab.py
PE: 192.168.0.123
Please specify pre-check file-name: Consolidated_BGP_file

Logged in SJ-PE | Model = MX240
Total Estab = 390 Down Peer = 5
Hang on.. Parsing data, might take few minutes
Data stored in file SJ-PE
Comparing Consolidated_BGP_file and SJ-PE
Difference output is stored in file 'Missing_in_new_PE'
```

The format in which data is stored in the files:

```
172.16.1.5 inet.0 Established
192.168.0.35 bgp.l3vpn.0 Established
192.168.0.20 bgp.l3vpn.0 Established
10.1.1.1 CUST1.inet.0 Established
10.26.1.1 CUST26.inet.0 Established
```

You can manipulate the format in which data is stored in files as per your preference.

TIP The above pre-check and post-check scripts can be copied and pasted.

Discussion

These scripts can ease the pain in verifying BGP states during the integration of two PEs and can inform the engineer if something is wrong with BGP states that don't match with pre-checks. Juniper has a great tool for one-on-one pre- and post-checks called *JSNAPy*. JSNAPy can be used when you are making changes on a specific device and need to compare its before and after status.

MORE? For more information on JSNAPy, there's an excellent *Day One* book: *Enabling Automated Network Verifications with JSNAPy*, available here: https://www.juniper.net/uk/en/training/jnbooks/day-one/automation-series/jsnapy/.

Recipe 8: L2VPN to VPLS Stitching

By Paul Clarke

- Junos OS used: 12.3X48-D65.1
- Juniper Platforms General Applicability: SRX

This recipe shows you how to stitch a Layer 2 Virtual Private Network (L2VPN) and a Virtual Private LAN Service (VPLS) together using logical tunnel interfaces: first the L2VPN and VPLS configurations, and then the stitching.

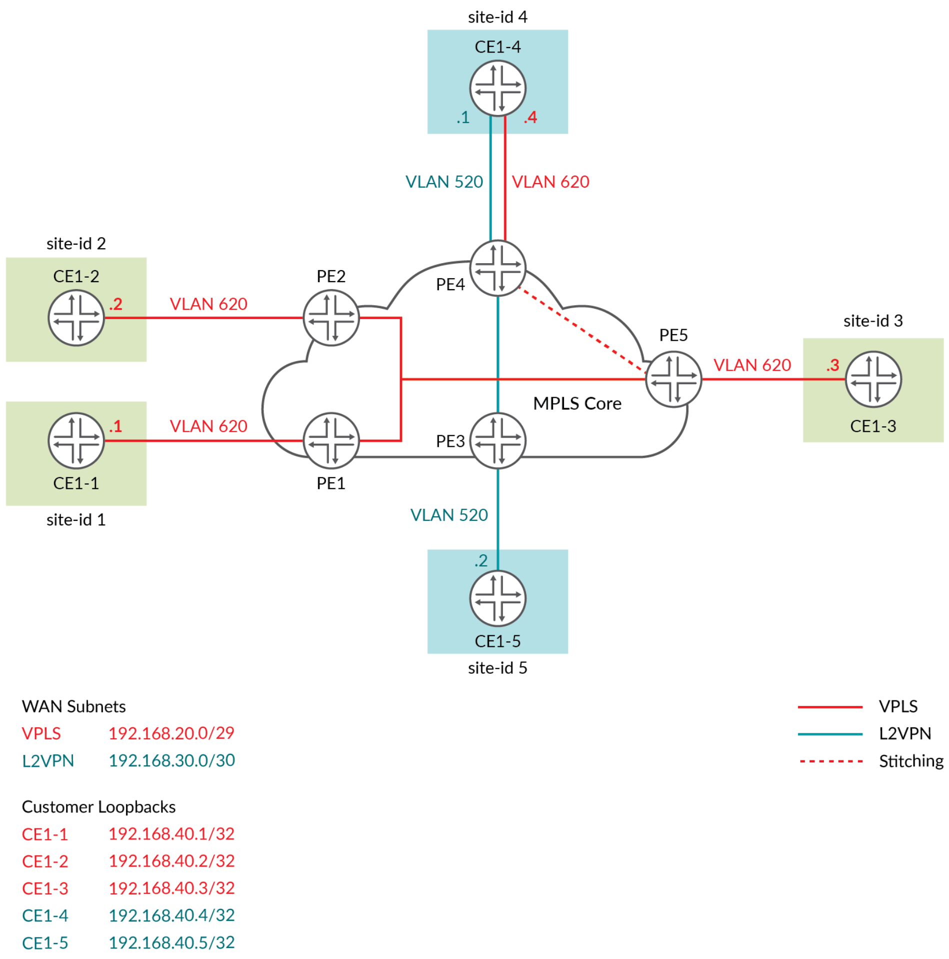
A Layer 2 VPN provides complete separation between the provider's network and the customer's network - that is, the PE devices and the CE devices do not exchange routing information. Some benefits of a Layer 2 VPN are that it is private, secure, and flexible. While VPLS is an Ethernet-based point-to-multipoint Layer 2 VPN, it allows you to connect geographically dispersed Ethernet LAN sites to each other across an MPLS backbone. For networks that implement VPLS, all sites appear to be in the same Ethernet LAN even though traffic travels across the service provider's network.

Problem

You have a requirement to interconnect your L2VPN service to your VPLS service without redesigning and reconfiguring the entire network.

Solution

L2VPN to VPLS stitching offers a neat solution without disruption to your or your customers' networks. Figure 8.1 illustrates the overall network topology with CE1-1, CE1-2, and CE1-3 participating in a VPLS segment.



WAN Subnets

VPLS	192.168.20.0/29
L2VPN	192.168.30.0/30

Customer Loopbacks

CE1-1	192.168.40.1/32
CE1-2	192.168.40.2/32
CE1-3	192.168.40.3/32
CE1-4	192.168.40.4/32
CE1-5	192.168.40.5/32

Figure 8.1 *Recipe 8 Network Topology*

In order to stitch the two services together, CE1-4 attached to provider edge (PE) router PE4 must be connected over the BGP L2VPN routing instance and connect to the VPLS routing instance configured on router PE5, while CE1-4 and CE1-5 have a point-to-point L2VPN service configured. The two services will be stitched together between PE4 and PE5.

First let's take a look at the PE interface and routing instance configuration for each of the VPLS-connected CE routers. The only real configuration that changes across the three PE routers will be the site and the site-identifier. Note that the interface number may also differ from PE to PE:

```
PE1#show interfaces ge-0/0/4
description "CE1-1 VPLS Router";
vlan-tagging;
encapsulation flexible-ethernet-services;
unit 620 {
    encapsulation vlan-vpls;
    vlan-id 620;
}

PE1# show routing-instances
VPN1-VPLS {
    instance-type vpls;
    vlan-id all;
    interface ge-0/0/4.620;
    vrf-target target:6779:105;
    protocols {
        vpls {
            site-range 3;
            no-tunnel-services;
            site CE1-1 {
                site-identifier 1;
            }
        }
    }
}
```

PE2 and PE5 will have a similar configuration. Use the `show vpls connections` command to show the status of the VPLS connections:

```
PE1# run show vpls connections
Layer-2 VPN connections:

Legend for connection status (St)
EI -- encapsulation invalid      NC -- interface encapsulation not CCC/TCC/VPLS
EM -- encapsulation mismatch     WE -- interface and instance encaps not same
VC-Dn -- Virtual circuit down    NP -- interface hardware not present
CM -- control-word mismatch      -> -- only outbound connection is up
CN -- circuit not provisioned    <- -- only inbound connection is up
OR -- out of range               Up -- operational
OL -- no outgoing label          Dn -- down
LD -- local site signaled down   CF -- call admission control failure
RD -- remote site signaled down  SC -- local and remote site ID collision
LN -- local site not designated  LM -- local site ID not minimum designated
RN -- remote site not designated RM -- remote site ID not minimum designated
XX -- unknown connection status  IL -- no incoming label
MM -- MTU mismatch               MI -- Mesh-Group ID not available
BK -- Backup connection          ST -- Standby connection
PF -- Profile parse failure      PB -- Profile busy
RS -- remote site standby        SN -- Static Neighbor
LB -- Local site not best-site   RB -- Remote site not best-site
VM -- VLAN ID mismatch

Legend for interface status
Up -- operational
Dn -- down
```

```
Instance: VPN1-VPLS
  Local site: CE1-1 (1)
    connection-site           Type  St     Time last up          # Up trans
    2                         rmt   Up     Dec  4 15:44:38 2018           1
      Remote PE: 172.27.255.2, Negotiated control-word: No
      Incoming label: 262146, Outgoing label: 262153
      Local interface: lsi.1048837, Status: Up, Encapsulation: VPLS
        Description: Intf - vpls VPN1-VPLS local site 1 remote site 2
    3                         rmt   Up     Dec  4 15:44:03 2018           1
      Remote PE: 172.27.255.5, Negotiated control-word: No
      Incoming label: 262147, Outgoing label: 262153
      Local interface: lsi.1048835, Status: Up, Encapsulation: VPLS
        Description: Intf - vpls VPN1-VPLS local site 1 remote site 3
```

When all three PE routers are configured and the connected CE routers are provisioned to run the OSPF routing protocol, connectivity between all three sites is achieved. The topology remains isolated from the L2VPN at this point.

CE1-1 now has OSPF adjacencies to the other two connected VPLS routers and can ping the loopback addresses, advertised by OSPF, of the remote CE devices CE1-2 and CE1-3 as shown here:

```
Virtual-Router# run show ospf neighbor instance CE1-1
Address          Interface              State     ID               Pri  Dead
192.168.20.3     ge-0/0/0.620           Full      192.168.40.3     128    33
192.168.20.2     ge-0/0/0.620           Full      192.168.40.2     128    37

Virtual-Router# run ping routing-instance CE1-1 192.168.40.2 rapid
PING 192.168.40.2 (192.168.40.2): 56 data bytes
!!!!!
--- 192.168.40.2 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.767/0.847/0.977/0.075 ms
Virtual-Router# run ping routing-instance CE1-1 192.168.40.3 rapid
PING 192.168.40.3 (192.168.40.3): 56 data bytes
!!!!!
--- 192.168.40.3 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max/stddev = 2.148/4.522/7.784/2.067 ms
```

Now let's have a look at the configuration requirements for the point-to-point L2VPN on PE3 and PE4. The PE configuration across the two PE routers will largely be the same apart from the site, site-identifier, and the remote site ID. The interface number may also differ from PE to PE:

```
PE3# show interfaces ge-0/0/3
description "CE1-5 L2VPN Router";
vlan-tagging;
encapsulation flexible-ethernet-services;
unit 520 {
    encapsulation vlan-ccc;
    vlan-id 520;
}
```

```
PE3# show routing-instances
VPN1-L2VPN {
    instance-type l2vpn;
    interface ge-0/0/3.520;
    vrf-target target:6779:100;
    protocols {
        l2vpn {
            encapsulation-type ethernet-vlan;
            interface ge-0/0/3.520;
            site CE1-5 {
                site-identifier 5;
                interface ge-0/0/3.520 {
                    remote-site-id 4;
                }
            }
        }
    }
}
```

PE4 will have a similar configuration. Use the `show l2vpn connections` command to show the status of the L2VPN connection:

```
PE3# run show l2vpn connections
Layer-2 VPN connections:

Legend for connection status (St)
EI -- encapsulation invalid      NC -- interface encapsulation not CCC/TCC/VPLS
EM -- encapsulation mismatch     WE -- interface and instance encaps not same
VC-Dn -- Virtual circuit down    NP -- interface hardware not present
CM -- control-word mismatch      -> -- only outbound connection is up
CN -- circuit not provisioned    <- -- only inbound connection is up
OR -- out of range               Up -- operational
OL -- no outgoing label          Dn -- down
LD -- local site signaled down   CF -- call admission control failure
RD -- remote site signaled down  SC -- local and remote site ID collision
LN -- local site not designated  LM -- local site ID not minimum designated
RN -- remote site not designated RM -- remote site ID not minimum designated
XX -- unknown connection status  IL -- no incoming label
MM -- MTU mismatch               MI -- Mesh-Group ID not available
BK -- Backup connection          ST -- Standby connection
PF -- Profile parse failure      PB -- Profile busy
RS -- remote site standby        SN -- Static Neighbor
LB -- Local site not best-site   RB -- Remote site not best-site
VM -- VLAN ID mismatch

Legend for interface status
Up -- operational
Dn -- down

Instance: VPN1-L2VPN
  Local site: CE1-5 (5)
    connection-site           Type  St     Time last up          # Up trans
    4                         rmt   Up     Dec  4 15:43:14 2018           1
      Remote PE: 172.27.255.4, Negotiated control-word: Yes (Null)
      Incoming label: 800001, Outgoing label: 800000
      Local interface: ge-0/0/3.520, Status: Up, Encapsulation: VLAN
```

When both PE routers are configured and the connected CE routers are provisioned to run OSPF, then connectivity between both sites is achieved. The topology remains isolated from the VPLS routers at this point.

CE1-5 now has an OSPF adjacency to CE1-4 and can ping the loopback address, advertised by OSPF, of the remote CE as shown here:

```
Virtual-Router# run show ospf neighbor instance CE1-5
Address          Interface              State     ID               Pri  Dead
192.168.30.1     ge-0/0/2.520           Full      192.168.40.4     128    37

Virtual-Router# run ping routing-instance CE1-5 192.168.40.4 rapid
PING 192.168.40.4 (192.168.40.4): 56 data bytes
!!!!!
--- 192.168.40.4 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.729/0.856/0.991/0.108 ms
```

Now for the clever part – and the recipe's "ah-ha" moment – stitching the two routing instances together.

The configuration on router PE4 must be amended to extend the L2VPN to PE5. This is done by adding a new unit to the exiting interface using the same VLAN as the VPLS instance - VLAN 620:

```
PE4#show interfaces ge-0/0/4
description "CE1-4 L2VPN Router";
vlan-tagging;
encapsulation flexible-ethernet-services;
unit 520 {
    encapsulation vlan-ccc;
    vlan-id 520;
}
unit 620 {
    encapsulation vlan-ccc;
    vlan-id 620;
}
```

The new interface and remote site ID, in this case 3, needs to be added to the existing L2VPN routing instance as shown here:

```
PE4# show routing-instances
VPN1-L2VPN {
    instance-type l2vpn;
    interface ge-0/0/4.520;
    interface ge-0/0/4.620;
    vrf-target target:6779:100;
    protocols {
        l2vpn {
            encapsulation-type ethernet-vlan;
            interface ge-0/0/4.520;
            site CE1-4 {
                site-identifier 4;
                interface ge-0/0/4.520 {
                    remote-site-id 5;
                }
                interface ge-0/0/4.620 {
                    remote-site-id 3;
                }
            }
        }
    }
}
```

Stitching the two services together takes place on the PE router PE5 by connecting the two routing instances together using a logical connection.

To connect two routing instances with a logical connection, configure a logical tunnel interface for each instance. Then, configure a peer relationship between the logical tunnel interfaces, thus creating a point-to-point connection. To configure a point-to-point connection between two routing instances, configure the logical tunnel interface using the `lt-fpc/pic/port` format.

On PE5 create two logical tunnel interfaces and peer them directly with each other. One unit will be configured for `vlan-ccc` and the other for `vlan-vpls` as shown next. Both units will be configured to use VLAN 620:

```
PE5# show interfaces lt-0/0/0
unit 0 {
    encapsulation vlan-ccc;
    vlan-id 620;
    peer-unit 1;
}
unit 1 {
    encapsulation vlan-vpls;
    vlan-id 620;
    peer-unit 0;
}
```

Create a new routing instance and place interface lt-0/0/0.0 in the L2VPN and configure a remote connection to PE4 using remote-site-id 4:

```
PE5# show routing-instances
VPN1-L2VPN {
    instance-type l2vpn;
    interface lt-0/0/0.0;
    vrf-target target:6779:100;
    protocols {
        l2vpn {
            encapsulation-type ethernet-vlan;
            interface lt-0/0/0.0;
            site CE1-3 {
                site-identifier 3;
                interface lt-0/0/0.0 {
                    remote-site-id 4;
                }
            }
        }
    }
}
```

To complete the stitching now add interface lt-0/0/0.1 to the existing VPLS routing instance already configured on router PE5 as shown here:

```
PE5# show routing-instances
VPN1-VPLS {
    instance-type vpls;
    vlan-id all;
    interface lt-0/0/0.1;
    interface ge-0/0/3.620;
    vrf-target target:6779:105;
    protocols {
        vpls {
            site-range 3;
            no-tunnel-services;
            site CE1-3 {
                site-identifier 3;
            }
        }
    }
}
```

Router CE1-3 now has OSPF adjacencies to CE1-1, CE1-2, and CE1-4. Because all routers are participating in OSPF and advertising their loopback addresses, CE1-3 can ping all routers in the topology:

```
Virtual-Router# run show ospf neighbor instance CE1-3
Address          Interface              State     ID               Pri  Dead
192.168.20.4     ge-0/0/4.620           Full      192.168.40.4     128    38
192.168.20.2     ge-0/0/4.620           Full      192.168.40.2     128    31
192.168.20.1     ge-0/0/4.620           Full      192.168.40.1     128    33
```

Now let’s verify our work. All routes are learned by router CE1-3:

```
Virtual-Router# run show route table CE1-3.inet.0

CE1-3.inet.0: 12 destinations, 12 routes (12 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

192.168.20.0/29    *[Direct/0] 1d 01:41:53
                    > via ge-0/0/4.620
192.168.20.1/32    *[OSPF/10] 1d 01:03:02, metric 1
                    > to 192.168.20.1 via ge-0/0/4.620
192.168.20.2/32    *[OSPF/10] 1d 01:02:16, metric 1
                    > to 192.168.20.2 via ge-0/0/4.620
192.168.20.3/32    *[Local/0] 2d 17:49:40
                      Local via ge-0/0/4.620
192.168.20.4/32    *[OSPF/10] 1d 01:00:38, metric 1
                    > to 192.168.20.4 via ge-0/0/4.620
192.168.30.0/30    *[OSPF/10] 1d 01:00:38, metric 2
                    > to 192.168.20.4 via ge-0/0/4.620
192.168.40.1/32    *[OSPF/10] 1d 01:03:02, metric 1
                    > to 192.168.20.1 via ge-0/0/4.620
192.168.40.2/32    *[OSPF/10] 1d 01:02:16, metric 1
                    > to 192.168.20.2 via ge-0/0/4.620
192.168.40.3/32    *[Direct/0] 2d 17:57:11
                    > via lo0.3
192.168.40.4/32    *[OSPF/10] 1d 01:00:38, metric 1
                    > to 192.168.20.4 via ge-0/0/4.620
```

```
192.168.40.5/32    *[OSPF/10] 1d 01:00:38, metric 2
                    > to 192.168.20.4 via ge-0/0/4.620
224.0.0.5/32       *[OSPF/10] 2d 17:45:50, metric 1
                      MultiRecv
```

And verify that all loopback addresses are reachable:

```
Virtual-Router# run ping routing-instance CE1-3 192.168.40.1 rapid
PING 192.168.40.1 (192.168.40.1): 56 data bytes
!!!!!
--- 192.168.40.1 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.866/0.941/1.078/0.090 ms

Virtual-Router# run ping routing-instance CE1-3 192.168.40.2 rapid
PING 192.168.40.2 (192.168.40.2): 56 data bytes
!!!!!
--- 192.168.40.2 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.980/1.012/1.075/0.034 ms

Virtual-Router# run ping routing-instance CE1-3 192.168.40.4 rapid
PING 192.168.40.4 (192.168.40.4): 56 data bytes
!!!!!
--- 192.168.40.4 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.758/0.870/1.049/0.098 ms

Virtual-Router# run ping routing-instance CE1-3 192.168.40.5 rapid
PING 192.168.40.5 (192.168.40.5): 56 data bytes
!!!!!
--- 192.168.40.5 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max/stddev = 1.104/1.206/1.369/0.092 ms
```

Discussion

As the need to link different Layer 2 services to one another for expanded service offerings grows, Layer 2 MPLS VPN services are increasingly in demand.

As with most networking solutions, there are always different ways to achieve the same goal. For instance, it is also possible to stitch services together using interworking interfaces. To configure the interworking interface, include the iw0 statement.

In addition to configuring the interworking interface, the l2iw protocol has to be enabled. This is a simple task using the set protocols l2iw command.

Another way to do this is to use a Tunnel Services PIC to loop packets out and back from the Packet Forwarding Engine to link together Layer 2 networks. The Layer 2 interworking software interface avoids the need for the Tunnel Services PIC and overcomes the limitation of bandwidth constraints imposed by the Tunnel Services PIC.

Recipe 9: Configuring EVPN VLAN-Aware Bundle Service on Juniper MX

By Dan Hearty

- Junos OS Used: 18.4R1.8
- Juniper Platforms General Applicability: MX Series Routers

Ethernet VPN (EVPN) technology was initially developed to overcome some of the inherent limitations of VPLS. Such limitations include multihoming and redundancy (single-active), inefficient flooding for MAC learning, and a lack of Layer 3 support. EVPN helps solve these problems and serves as a great solution for interconnecting Layer 2 domains, among many other benefits.

In the control plane, EVPN leverages BGP via a dedicated network layer reachability information (NLRI). In the data plane, a number of encapsulation types are supported, however VXLAN and MPLS remain the most popular options. More specifically, VXLAN is a popular choice within the data center and serves as a good solution for Data Center Interconnect (DCI). MPLS, on the other hand, remains very popular in the service provider world.

EVPN currently offers three primary service types: VLAN based, VLAN-bundle, and VLAN-aware bundle. Each service type provides various benefits. This recipe focuses specifically on *VLAN-aware bundle* service, which allows multiple VLANs to map to a single EVPN instance. Each VLAN within the EVI maps to a dedicated bridge domain, meaning each VLAN benefits from its own dedicated broadcast domain. This differs somewhat from the VLAN bundle service, whereby a single broadcast domain is shared by all VLANs connected to the EVI. EVPN VLAN-aware bundle supports VLAN normalization and overlapping MAC addresses.

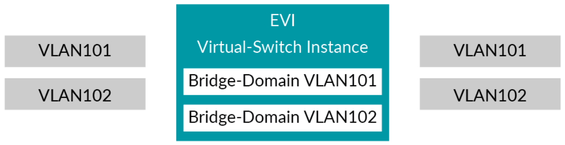


Figure 9.1 *VLAN-Aware Bundle*

Problem

In this scenario there are two customer sites, C1-1 and C1-2, that are connected via an MPLS-enabled core network. Multiple VLANs are in use at each site and there is a requirement to provide Layer 2 connectivity between the sites. EVPN must be used and each VLAN should be mapped to its own dedicated broadcast domain. However, only a single EVI should be used in the solution. Customer site C1-2 VLAN1222 should be able to communicate with site C1-1 VLAN102. Lastly, there are overlapping MAC addresses present and this should not have an adverse effect.

Solution

To solve this problem, an EVPN VLAN-aware bundle service will be configured on MX Series devices to provide connectivity between the customer site C1-1 and C1-2. EVPN VLAN-aware bundle service supports per-VLAN broadcast domains that are all contained within a single EVPN instance. This means that overlapping MAC addressing does not cause any adverse behavior. VLAN normalization techniques will be used to enable traffic flow between C1-1 VLAN102 and C1-2 VLAN1222.

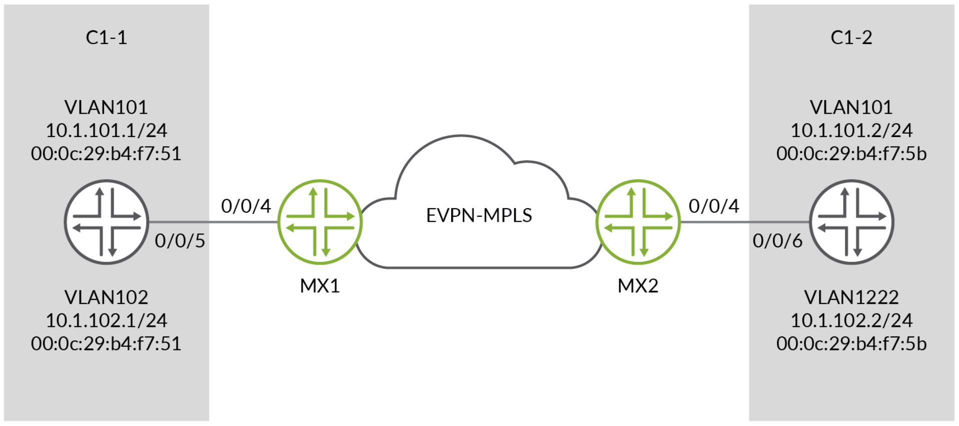


Figure 9.2 Topology for Recipe 9's VLAN-Aware Bundle

Overview

This recipe details the steps required to configure and verify an EVPN VLAN-aware bundle service between Juniper vMX devices, enabling the core for IGP, MPLS, and BGP.

Configure the CE

In this example, a single vSRX is used to simulate customer site C1-1 and C1-2. Virtual Routers are used to separate control plane and data plane functions between each site. For more information on how to configure routing instances, refer to this book's Recipe #1: *Virtualizing Routers with Routing Instances*, by Martin Brown.

Configure CE Interfaces

We'll kick things off by configuring the interfaces that are used by C1-1 and C1-2 routing instances. Each interface is configured with two 802.1Q tagged sub-interfaces with `vlan-tagging` enabled. Each sub-interface is assigned an IP address within its associated VLANs. We'll use these later for verification:

```
lab@vSRX-VR> show configuration interfaces
ge-0/0/5 {
    vlan-tagging;
    unit 101 {
        vlan-id 101;
        family inet {
            address 10.1.101.1/24;
        }
    }
    unit 102 {
        vlan-id 102;
        family inet {
            address 10.1.102.1/24;
        }
    }
}
ge-0/0/6 {
    unit 101 {
        vlan-id 101;
        family inet {
            address 10.1.101.2/24;
        }
    }
    unit 102 {
        vlan-id 1222;
        family inet {
            address 10.1.102.2/24;
        }
    }
}

lab@vSRX-VR> show configuration interfaces | display set
set interfaces ge-0/0/5 vlan-tagging
set interfaces ge-0/0/5 unit 101 vlan-id 101
set interfaces ge-0/0/5 unit 101 family inet address 10.1.101.1/24
set interfaces ge-0/0/5 unit 102 vlan-id 102
set interfaces ge-0/0/5 unit 102 family inet address 10.1.102.1/24
set interfaces ge-0/0/6 vlan-tagging
set interfaces ge-0/0/6 unit 101 vlan-id 101
set interfaces ge-0/0/6 unit 101 family inet address 10.1.101.2/24
set interfaces ge-0/0/6 unit 102 vlan-id 1222
set interfaces ge-0/0/6 unit 102 family inet address 10.1.102.2/24
```

Configure C1-1 and C1-2 Routing Instances

Now create two virtual router routing instances named C1-1 & C1-2 and assign the interfaces, configured in the previous step, to the relevant `routing-instances`:

```
lab@vSRX-VR> show configuration routing-instances
CUST1-1 {
    instance-type virtual-router;
    interface ge-0/0/5.101;
    interface ge-0/0/5.102;
}
CUST1-2 {
    instance-type virtual-router;
    interface ge-0/0/6.101;
    interface ge-0/0/6.102;

lab@vSRX-VR> show configuration routing-instances | display set
set routing-instances CUST1-1 instance-type virtual-router
set routing-instances CUST1-1 interface ge-0/0/5.101
set routing-instances CUST1-1 interface ge-0/0/5.102
set routing-instances CUST1-2 instance-type virtual-router
set routing-instances CUST1-2 interface ge-0/0/6.101
set routing-instances CUST1-2 interface ge-0/0/6.102
```

Configure MX1

The following section details all the necessary steps required to configure IGP, MPLS, BGP, and an EVPN VLAN-aware bundle service on MX1. You may want to skip to the last section, *Configure VLAN-Aware Bundle Service,* if you already have an MPLS core network enabled.

Configure IGP

ISIS is used in the IGP to provide loopback connectivity between MX1 and MX2.

NOTE Ensure family ISO is enabled on core facing interfaces and also ensure a valid ISO address is configured on lo0:

```
lab@vMX1> show configuration protocols
isis {
    interface ge-0/0/0.11 {
        point-to-point;
    }
    interface lo0.0;
}

lab@vMX1> show configuration protocols isis | display set
set protocols isis interface ge-0/0/0.11 point-to-point
set protocols isis interface lo0.0
```

Configure MPLS

MPLS is enabled on core facing interfaces.

NOTE Ensure family MPLS is enabled on core facing interfaces:

```
lab@vMX1> show configuration protocols
mpls {
    interface ge-0/0/0.11;

lab@vMX1> show configuration protocols mpls | display set
set protocols mpls interface ge-0/0/0.11
```

Configure LDP

LDP is configured for lo0 and the core facing interface:

```
lab@vMX1> show configuration protocols
ldp {
    interface ge-0/0/0.11;
    interface lo0.0;

lab@vMX1> show configuration protocols ldp | display set
set protocols ldp interface ge-0/0/0.11
set protocols ldp interface lo0.0
```

Configure BGP

Now that we've enabled loopback reachability between MX1 and MX2, and LDP has been enabled, BGP can be configured. A group named `EVPN` is created with an EBGP peering to MX2. The `multihop` knob is required as the TTL is greater than 1 to reach MX2. Note that `family evpn signaling` is configured for the peering. This enables the exchange of EVPN information only and overrides the default family inet setting:

```
lab@vMX1> show configuration protocols
bgp {
    group EVPN {
        type external;
        multihop;
        local-address 10.0.255.21;
        family evpn {
            signaling;
        }
        local-as 65101;
        neighbor 10.0.255.23 {
            peer-as 65001;
        }
    }
}

lab@vMX1> show configuration protocols bgp | display set
set protocols bgp group EVPN type external
set protocols bgp group EVPN multihop
set protocols bgp group EVPN local-address 10.0.255.21
set protocols bgp group EVPN family evpn signaling
set protocols bgp group EVPN local-as 65101
set protocols bgp group EVPN neighbor 10.0.255.23 peer-as 65001
```

Configure CE-Facing Interface

EVPN VLAN-aware bundle supports the use of trunk interfaces to connect the CE into the EVPN instance. The PE-CE interface is configured to accept VLAN-tagged frames from C1-1 VLAN101 and VLAN102:

```
lab@vMX1> show configuration interfaces ge-0/0/4
flexible-vlan-tagging;
encapsulation flexible-ethernet-services;
unit 0 {
    family bridge {
        interface-mode trunk;
        vlan-id-list [ 101 102 ];
    }
}
```

```
set interfaces ge-0/0/4 flexible-vlan-tagging
set interfaces ge-0/0/4 encapsulation flexible-ethernet-services
set interfaces ge-0/0/4 unit 0 family bridge interface-mode trunk
set interfaces ge-0/0/4 unit 0 family bridge vlan-id-list 101
set interfaces ge-0/0/4 unit 0 family bridge vlan-id-list 102
```

Configure VLAN-Aware Bundle Service

Configuring an EVPN VLAN-aware bundle involves creating a virtual switch routing instance, assigning PE-CE interfaces, and enabling EVPN. Bridge domains are configured on a per-VLAN basis and you must list those VLANs via `extended-vlan-list` under protocols EVPN. Also, as we are using MPLS for transport, there must be a `route-distinguisher` and `vrf-target` assigned for the service.

NOTE EVPN VLAN-aware bundle uses virtual switch routing instances. Bridge domains are configured on a per-VLAN basis to provide separate broadcast domains within the routing instance:

```
lab@vMX1> show configuration routing-instances
CUST1 {
    instance-type virtual-switch;
    interface ge-0/0/4.0;
    route-distinguisher 10.0.255.21:1;
    vrf-target target:65000:100;
    protocols {
        evpn {
            extended-vlan-list 101-102;
        }
    }
    bridge-domains {
        VLAN101 {
            vlan-id 101;
        }
        VLAN102 {
            vlan-id 102;
        }
    }
}

lab@vMX1> **show configuration routing-instances | display set**
set routing-instances CUST1 instance-type virtual-switch
set routing-instances CUST1 interface ge-0/0/4.0
set routing-instances CUST1 route-distinguisher 10.0.255.21:1
set routing-instances CUST1 vrf-target target:65000:100
set routing-instances CUST1 protocols evpn extended-vlan-list 101-102
set routing-instances CUST1 bridge-domains VLAN101 vlan-id 101
set routing-instances CUST1 bridge-domains VLAN102 vlan-id 102
```

Configure MX2

The following section is very similar to the previous one, although this time we focus on MX2. Once again, you may want to skip to the last section, *Configure VLAN-Aware Bundle Service,* if you already have an MPLS core network enabled.

Configure IGP

ISIS is used in the IGP to provide loopback connectivity between MX2 and MX1.

NOTE Ensure family ISO is enabled on core facing interfaces and also ensure a valid ISO address is configured on lo0:

```
lab@vMX2> show configuration protocols
isis {
    interface ge-0/0/3.0 {
        point-to-point;
    }
    interface lo0.0;
}

lab@vMX2> show configuration protocols isis | display set
set protocols isis interface ge-0/0/3.0 point-to-point
set protocols isis interface lo0.0
```

Configure MPLS

MPLS is enabled on core facing interfaces.

NOTE Ensure family MPLS is enabled on core facing interfaces:

```
lab@vMX2> show configuration protocols
mpls {
    interface ge-0/0/3.0;
}
```

```
lab@vMX2> show configuration protocols mpls | display set
set protocols mpls interface ge-0/0/3.0
```

Configure LDP

LDP is configured for lo0 and the core facing interface:

```
lab@vMX2> show configuration protocols
ldp {
    interface ge-0/0/3.0;
    interface lo0.0;
}

lab@vMX2> show configuration protocols ldp | display set
set protocols ldp interface ge-0/0/3.0
set protocols ldp interface lo0.0
```

Configure BGP

Now that we've enabled loopback reachability between MX2 and MX1, and LDP has been enabled, BGP can be configured. A group named `EVPN` is created with an EBGP peering to MX1. The `multihop` knob is required as the TTL is greater than 1 to reach MX1. Note that `family evpn signaling` is configured for the peering. This enables the exchange of EVPN information only and overrides the default family inet setting:

```
lab@vMX2> show configuration protocols
bgp {
    group EVPN {
        type external;
        multihop;
        local-address 10.0.255.23;
        family evpn {
            signaling;
        }
        local-as 65001;
        neighbor 10.0.255.21 {
            peer-as 65101;
        }
    }
}

lab@vMX2> show configuration protocols bgp | display set
set protocols bgp group EVPN type external
set protocols bgp group EVPN multihop
set protocols bgp group EVPN local-address 10.0.255.23
set protocols bgp group EVPN family evpn signaling
set protocols bgp group EVPN local-as 65001
set protocols bgp group EVPN neighbor 10.0.255.21 peer-as 65101
```

Configure CE-Facing Interface

EVPN VLAN-aware bundle supports the use of trunk interfaces to connect the CE into the EVPN instance. The PE-CE interface is configured to accept VLAN tagged frames from C1-1 VLAN101. There is a requirement to enable communication between C1-1 VLAN102 and C1-2 VLAN1222. VLAN normalization techniques are used on the interface to translate VLAN1222 to VLAN102, as shown in Figure 9.3. VLAN102 is then used within the EVPN service, thus providing connectivity between C1-1 VLAN102 and C1-2 VLAN1222:

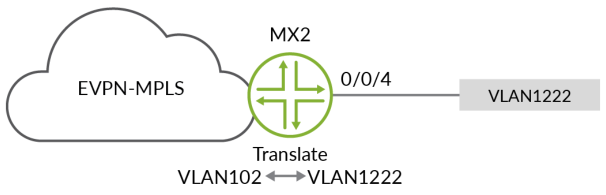


Figure 9.3 VLAN-Aware Normalization

```
lab@vMX2> show configuration interfaces ge-0/0/4
flexible-vlan-tagging;
encapsulation flexible-ethernet-services;
unit 0 {
    family bridge {
        interface-mode trunk;
        vlan-id-list [ 101 102 ];
        vlan-rewrite {
            translate 1222 102;
        }
    }
}

lab@vMX2> show configuration interfaces ge-0/0/4 | display set
set interfaces ge-0/0/4 flexible-vlan-tagging
set interfaces ge-0/0/4 encapsulation flexible-ethernet-services
set interfaces ge-0/0/4 unit 0 family bridge interface-mode trunk
set interfaces ge-0/0/4 unit 0 family bridge vlan-id-list 101
set interfaces ge-0/0/4 unit 0 family bridge vlan-id-list 102
set interfaces ge-0/0/4 unit 0 family bridge vlan-rewrite translate 1222 102
```

Configure VLAN-Aware Bundle Service

Configuring an EVPN VLAN-aware bundle involves creating a virtual switch routing instance, assigning PE-CE interfaces, and enabling EVPN. Bridge domains are configured on a per VLAN basis and you must list those VLANs via `extended-vlan-list` under protocols EVPN. Note that we use the translated VLAN102, which was configured in the previous step, as opposed to the original VLAN1222. Also, as we are using MPLS for transport, there must be a `route-distinguisher` and `vrf-target` assigned for the service.

NOTE The EVPN VLAN-aware bundle uses virtual switch routing instances. Bridge domains are configured on a per VLAN basis to provide separate broadcast domains within the routing instance:

```
lab@vMX2> show configuration routing-instances
CUST1 {
    instance-type virtual-switch;
    interface ge-0/0/4.0;
    route-distinguisher 10.0.255.23:1;
    vrf-target target:65000:100;
    protocols {
        evpn {
            extended-vlan-list 101-102;
        }
    }
    bridge-domains {
        VLAN101 {
            vlan-id 101;
        }
        VLAN102 {
            vlan-id 102;
        }
    }
}

lab@vMX2> show configuration routing-instances | display set
set routing-instances CUST1 instance-type virtual-switch
set routing-instances CUST1 interface ge-0/0/4.0
set routing-instances CUST1 route-distinguisher 10.0.255.23:1
set routing-instances CUST1 vrf-target target:65000:100
set routing-instances CUST1 protocols evpn extended-vlan-list 101-102
set routing-instances CUST1 bridge-domains VLAN101 vlan-id 101
set routing-instances CUST1 bridge-domains VLAN102 vlan-id 102
```

Verification

The following section details the steps required to verify IGP, LDP, BGP, and the EVPN VLAN-aware bundle service.

IGP, LDP

First, verify that ISIS and LDP have been established with the core network and also ensure there is an MPLS label to reach the loopback of the remote MX device:

MX1

```
lab@vMX1> show isis adjacency
Interface             System         L State        Hold (secs) SNPA
ge-0/0/0.11           vSRX1          3  Up                   20


lab@vMX1> show ldp neighbor
Address                            Interface       Label space ID     Hold time
10.41.11.1                         ge-0/0/0.11     10.0.255.11:0        12
```

```
lab@vMX1> show route 10.0.255.23/32


inet.0: 33 destinations, 33 routes (33 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both


10.0.255.23/32     *[IS-IS/15] 1w5d 10:10:46, metric 30
                    >  to 10.41.11.1 via ge-0/0/0.11

inet.3: 6 destinations, 6 routes (6 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

10.0.255.23/32     *[LDP/9] 1w5d 10:10:45, metric 1
                    >  to 10.41.11.1 via ge-0/0/0.11, Push 346272
```

MX2

```
lab@vMX2> show isis adjacency
Interface             System         L State        Hold (secs) SNPA
ge-0/0/3.0            vSRX3          3  Up                   24


lab@vMX2> show ldp neighbor
Address                            Interface       Label space ID     Hold time
10.43.4.1                          ge-0/0/3.0      10.0.255.13:0        13


lab@vMX2> show route 10.0.255.21/32

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

10.0.255.21/32     *[IS-IS/15] 1w5d 10:14:54, metric 30
                    >  to 10.43.4.1 via ge-0/0/3.0

inet.3: 6 destinations, 6 routes (6 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

10.0.255.21/32     *[LDP/9] 1w5d 10:15:04, metric 1
                    >  to 10.43.4.1 via ge-0/0/3.0, Push 352560
```

This output verifies that ISIS and LDP have been established with the core network. You can also see that a /32 route has been installed in inet.0 and inet.3 for the remote MX loopback. On MX1, LDP has assigned the label value 346272 to reach MX2. On MX2, LDP has assigned the label value 352560 to reach MX1.

BGP

Now that there's an entry in inet.0 for the remote loopback, and BGP can resolve via inet.3, we can now verify BGP.

In this scenario we're using BGP to carry EVPN information. This was enabled by the family EVPN `signaling` knob under the BGP configuration group:

MX1

```
lab@vMX1> show bgp neighbor 10.0.255.23
Peer: 10.0.255.23+179 AS 65001 Local: 10.0.255.21+65167 AS 65101
  Group: EVPN                  Routing-Instance: master
  Forwarding routing-instance: master
  Type: External    State: Established    Flags: <Sync>
  Last State: OpenConfirm   Last Event: RecvKeepAlive
  Last Error: None
  Options: <Multihop Preference LocalAddress AddressFamily PeerAS LocalAS Rib-group Refresh>
  Address families configured: evpn
  Local Address: 10.0.255.21 Holdtime: 90 Preference: 170 Local AS: 65101 Local System AS: 65000
  Number of flaps: 0
  Peer ID: 10.0.255.23     Local ID: 10.0.255.21       Active Holdtime: 90
  Keepalive Interval: 30         Group index: 0    Peer index: 0    SNMP index: 0
  I/O Session Thread: bgpio-0 State: Enabled
  BFD: disabled, down
  NLRI for restart configured on peer: evpn
  NLRI advertised by peer: evpn
  NLRI for this session: evpn
  Peer supports Refresh capability (2)
  Stale routes from peer are kept for: 300
  Peer does not support Restarter functionality
  Restart flag received from the peer: Notification
  NLRI that restart is negotiated for: evpn
  NLRI of received end-of-rib markers: evpn
  NLRI of all end-of-rib markers sent: evpn
  Peer does not support LLGR Restarter functionality
  Peer supports 4 byte AS extension (peer-as 65001)
  Peer does not support Addpath
  Table CUST1.evpn.0
    RIB State: BGP restart is complete
    RIB State: VPN restart is complete
    Send state: not advertising
    Active prefixes:              5
    Received prefixes:            5
    Accepted prefixes:            5
    Suppressed due to damping:    0
  Table __default_evpn__.evpn.0
    RIB State: BGP restart is complete
    RIB State: VPN restart is complete
    Send state: not advertising
    Active prefixes:              0
    Received prefixes:            0
    Accepted prefixes:            0
    Suppressed due to damping:    0
  Table bgp.evpn.0 Bit: 20000
    RIB State: BGP restart is complete
    RIB State: VPN restart is complete
    Send state: in sync
    Active prefixes:              5
    Received prefixes:            5
    Accepted prefixes:            5
    Suppressed due to damping:    0
    Advertised prefixes:          5
  Last traffic (seconds): Received 10   Sent 6    Checked 1074958
  Input messages:  Total 41588  Updates 1926    Refreshes 0     Octets 984017
  Output messages: Total 41521  Updates 1704    Refreshes 0     Octets 961177
  Output Queue[1]: 0            (bgp.evpn.0, evpn)
```

MX2

```
lab@vMX2> show bgp neighbor 10.0.255.21
Peer: 10.0.255.21+65167 AS 65101 Local: 10.0.255.23+179 AS 65001
  Group: EVPN                  Routing-Instance: master
  Forwarding routing-instance: master
  Type: External    State: Established    Flags: <Sync>
  Last State: OpenConfirm   Last Event: RecvKeepAlive
  Last Error: None
  Options: <Multihop Preference LocalAddress AddressFamily PeerAS LocalAS Rib-group Refresh>
  Address families configured: evpn
  Local Address: 10.0.255.23 Holdtime: 90 Preference: 170 Local AS: 65001 Local System AS: 65000
  Number of flaps: 0
  Peer ID: 10.0.255.21     Local ID: 10.0.255.23       Active Holdtime: 90
  Keepalive Interval: 30         Group index: 1    Peer index: 0    SNMP index: 1
  I/O Session Thread: bgpio-0 State: Enabled
  BFD: disabled, down
  NLRI for restart configured on peer: evpn
  NLRI advertised by peer: evpn
  NLRI for this session: evpn
  Peer supports Refresh capability (2)
  Stale routes from peer are kept for: 300
  Peer does not support Restarter functionality
  Restart flag received from the peer: Notification
  NLRI that restart is negotiated for: evpn
  NLRI of received end-of-rib markers: evpn
  NLRI of all end-of-rib markers sent: evpn
  Peer does not support LLGR Restarter functionality
  Peer supports 4 byte AS extension (peer-as 65101)
  Peer does not support Addpath
  Table CUST1.evpn.0
    RIB State: BGP restart is complete
    RIB State: VPN restart is complete
    Send state: not advertising
    Active prefixes:              5
    Received prefixes:            5
    Accepted prefixes:            5
    Suppressed due to damping:    0
  Table __default_evpn__.evpn.0
    RIB State: BGP restart is complete
    RIB State: VPN restart is complete
    Send state: not advertising
    Active prefixes:              0
    Received prefixes:            0
    Accepted prefixes:            0
    Suppressed due to damping:    0
  Table bgp.evpn.0 Bit: 30000
    RIB State: BGP restart is complete
    RIB State: VPN restart is complete
    Send state: in sync
    Active prefixes:              5
    Received prefixes:            5
    Accepted prefixes:            5
    Suppressed due to damping:    0
    Advertised prefixes:          5
  Last traffic (seconds): Received 9    Sent 16   Checked 1074991
  Input messages:  Total 41524  Updates 1705    Refreshes 0     Octets 961278
  Output messages: Total 41589  Updates 1925    Refreshes 0     Octets 984036
  Output Queue[2]: 0            (bgp.evpn.0, evpn)
```

Confirmation that BGP has been successfully established between MX1 and MX3 and the session is enabled to carry EVPN information.

EVPN VLAN-Aware Bundle Service

The final step verifies that customer MAC addresses are imported into EVPN and exchanged between the MX devices.

The EVPN database provides information about locally learned MAC addresses and includes the interface on which the MAC was received. It also displays remotely learned MACs and includes the source, therefore the remote PE. You can also verify the time that the MAC entry was created/updated:

MX1

```
lab@vMX1> show evpn database
Instance: CUST1
VLAN  DomainId  MAC address        Active source                  Timestamp        IP address
101             00:0c:29:b4:f7:51  ge-0/0/4.0                     Feb 25 16:36:16  10.1.101.1
101             00:0c:29:b4:f7:5b  10.0.255.23                    Feb 25 16:40:29  10.1.101.2
102             00:0c:29:b4:f7:51  ge-0/0/4.0                     Feb 25 16:40:22  10.1.102.1
102             00:0c:29:b4:f7:5b  10.0.255.23                    Feb 25 16:22:50  10.1.102.2
```

MX2

```
lab@vMX2> show evpn database
Instance: CUST1
VLAN  DomainId  MAC address        Active source                  Timestamp        IP address
101             00:0c:29:b4:f7:51  10.0.255.21                    Feb 25 16:31:17  10.1.101.1
101             00:0c:29:b4:f7:5b  ge-0/0/4.0                     Feb 25 16:40:29  10.1.101.2
102             00:0c:29:b4:f7:51  10.0.255.21                    Feb 25 15:27:47  10.1.102.1
102             00:0c:29:b4:f7:5b  ge-0/0/4.0                     Feb 25 16:37:51  10.1.102.2
```

You can verify local and remote MAC addresses via the `bridge mac-table`. In this scenario there are two bridge MAC tables, one for each VLAN. Note that both bridge tables are members of the same `routing-instance`:

MX1

```
lab@vMX1> show bridge mac-table

MAC flags       (S -static MAC, D -dynamic MAC, L -locally learned, C -Control MAC
    O -OVSDB MAC, SE -Statistics enabled, NM -Non configured MAC, R -Remote PE MAC, P -Pinned MAC)

Routing instance : CUST1
 Bridging domain : VLAN101, VLAN : 101
   MAC                 MAC      Logical          NH     MAC         active
   address             flags    interface        Index  property    source
   00:0c:29:b4:f7:51   D        ge-0/0/4.0
   00:0c:29:b4:f7:5b   DC                        1048576            10.0.255.23

MAC flags       (S -static MAC, D -dynamic MAC, L -locally learned, C -Control MAC
    O -OVSDB MAC, SE -Statistics enabled, NM -Non configured MAC, R -Remote PE MAC, P -Pinned MAC)
```

```
Routing instance : CUST1
 Bridging domain : VLAN102, VLAN : 102
   MAC                 MAC      Logical          NH     MAC         active
   address             flags    interface        Index  property    source
   00:0c:29:b4:f7:51   D        ge-0/0/4.0
   00:0c:29:b4:f7:5b   DC                        1048576            10.0.255.23
```

MX2

```
lab@vMX2> show bridge mac-table

MAC flags       (S -static MAC, D -dynamic MAC, L -locally learned, C -Control MAC
    O -OVSDB MAC, SE -Statistics enabled, NM -Non configured MAC, R -Remote PE MAC, P -Pinned MAC)

Routing instance : CUST1
 Bridging domain : VLAN101, VLAN : 101
   MAC                 MAC      Logical          NH     MAC         active
   address             flags    interface        Index  property    source
   00:0c:29:b4:f7:51   DC                        1048574            10.0.255.21
   00:0c:29:b4:f7:5b   D        ge-0/0/4.0

MAC flags       (S -static MAC, D -dynamic MAC, L -locally learned, C -Control MAC
    O -OVSDB MAC, SE -Statistics enabled, NM -Non configured MAC, R -Remote PE MAC, P -Pinned MAC)

Routing instance : CUST1
 Bridging domain : VLAN102, VLAN : 102
   MAC                 MAC      Logical          NH     MAC         active
   address             flags    interface        Index  property    source
   00:0c:29:b4:f7:51   DC                        1048574            10.0.255.21
   00:0c:29:b4:f7:5b   D        ge-0/0/4.0
```

Once the locally learned MAC addresses have been imported into the local EVPN table, BGP is used to advertise details to the remote MX. BGP RIB-OUT for the CUST1 instance shows what's being announced to the remote MX. Note that each MAC is represented by two EVPN type2 routes. There is also an EVPN type3 route for each bridge domain. This is used to deal with BUM traffic:

MX1

```
lab@vMX1> show route advertising-protocol bgp 10.0.255.23 table CUST1.evpn.0

CUST1.evpn.0: 12 destinations, 12 routes (12 active, 0 holddown, 0 hidden)
  Prefix                  Nexthop              MED     Lclpref    AS path
  2:10.0.255.21:1::101::00:0c:29:b4:f7:51/304 MAC/IP
*                         Self                                    I
  2:10.0.255.21:1::102::00:0c:29:b4:f7:51/304 MAC/IP
*                         Self                                    I
  2:10.0.255.21:1::101::00:0c:29:b4:f7:51::10.1.101.1/304 MAC/IP
*                         Self                                    I
  2:10.0.255.21:1::102::00:0c:29:b4:f7:51::10.1.102.1/304 MAC/IP
*                         Self                                    I
  3:10.0.255.21:1::101::10.0.255.21/248 IM
*                         Self                                    I
  3:10.0.255.21:1::102::10.0.255.21/248 IM
*                         Self                                    I
```

MX2

```
lab@vMX2> show route advertising-protocol bgp 10.0.255.21 table CUST1.evpn.0

CUST1.evpn.0: 12 destinations, 12 routes (12 active, 0 holddown, 0 hidden)
  Prefix                  Nexthop              MED     Lclpref    AS path
  2:10.0.255.23:1::101::00:0c:29:b4:f7:5b/304 MAC/IP
*                         Self                                    I
  2:10.0.255.23:1::102::00:0c:29:b4:f7:5b/304 MAC/IP
*                         Self                                    I
  2:10.0.255.23:1::101::00:0c:29:b4:f7:5b::10.1.101.2/304 MAC/IP
*                         Self                                    I
  2:10.0.255.23:1::102::00:0c:29:b4:f7:5b::10.1.102.2/304 MAC/IP
*                         Self                                    I
  3:10.0.255.23:1::101::10.0.255.23/248 IM
*                         Self                                    I
  3:10.0.255.23:1::102::10.0.255.23/248 IM
*                         Self                                    I
```

Based on the `route-target` configured under the routing instance, EVPN routes are imported into the relevant instance. By checking the CUST1 routing table, you can see all the local and remote MACs that are participating in the EVPN VLAN-aware bundle service:

MX1

```
lab@vMX1> show route table CUST1.evpn.0

CUST1.evpn.0: 12 destinations, 12 routes (12 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

2:10.0.255.21:1::101::00:0c:29:b4:f7:51/304 MAC/IP
                   *[EVPN/170] 00:01:10
                       Indirect
2:10.0.255.21:1::102::00:0c:29:b4:f7:51/304 MAC/IP
                   *[EVPN/170] 00:01:02
                       Indirect
2:10.0.255.23:1::101::00:0c:29:b4:f7:5b/304 MAC/IP
                   *[BGP/170] 02:32:57, localpref 100, from 10.0.255.23
                      AS path: 65001 I, validation-state: unverified
                    >  to 10.41.11.1 via ge-0/0/0.11, Push 346272
2:10.0.255.23:1::102::00:0c:29:b4:f7:5b/304 MAC/IP
                   *[BGP/170] 00:01:02, localpref 100, from 10.0.255.23
                      AS path: 65001 I, validation-state: unverified
                    >  to 10.41.11.1 via ge-0/0/0.11, Push 346272
2:10.0.255.21:1::101::00:0c:29:b4:f7:51::10.1.101.1/304 MAC/IP
                   *[EVPN/170] 00:01:10
                       Indirect
2:10.0.255.21:1::102::00:0c:29:b4:f7:51::10.1.102.1/304 MAC/IP
                   *[EVPN/170] 00:01:02
                       Indirect
2:10.0.255.23:1::101::00:0c:29:b4:f7:5b::10.1.101.2/304 MAC/IP
                   *[BGP/170] 02:32:57, localpref 100, from 10.0.255.23
                      AS path: 65001 I, validation-state: unverified
                    >  to 10.41.11.1 via ge-0/0/0.11, Push 346272
2:10.0.255.23:1::102::00:0c:29:b4:f7:5b::10.1.102.2/304 MAC/IP
                   *[BGP/170] 00:01:02, localpref 100, from 10.0.255.23
```

```
                      AS path: 65001 I, validation-state: unverified
                    >  to 10.41.11.1 via ge-0/0/0.11, Push 346272
3:10.0.255.21:1::101::10.0.255.21/248 IM
                   *[EVPN/170] 3d 19:08:08
                       Indirect
3:10.0.255.21:1::102::10.0.255.21/248 IM
                   *[EVPN/170] 3d 19:08:08
                       Indirect
3:10.0.255.23:1::101::10.0.255.23/248 IM
                   *[BGP/170] 3d 19:05:41, localpref 100, from 10.0.255.23
                      AS path: 65001 I, validation-state: unverified
                    >  to 10.41.11.1 via ge-0/0/0.11, Push 346272
3:10.0.255.23:1::102::10.0.255.23/248 IM
                   *[BGP/170] 3d 19:05:41, localpref 100, from 10.0.255.23
                      AS path: 65001 I, validation-state: unverified
                    >  to 10.41.11.1 via ge-0/0/0.11, Push 346272
```

MX2

```
lab@vMX2> show route table CUST1.evpn.0

CUST1.evpn.0: 12 destinations, 12 routes (12 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

2:10.0.255.21:1::101::00:0c:29:b4:f7:51/304 MAC/IP
                   *[BGP/170] 00:43:16, localpref 100, from 10.0.255.21
                      AS path: 65101 I, validation-state: unverified
                    >  to 10.43.4.1 via ge-0/0/3.0, Push 352560
2:10.0.255.21:1::102::00:0c:29:b4:f7:51/304 MAC/IP
                   *[BGP/170] 00:33:54, localpref 100, from 10.0.255.21
                      AS path: 65101 I, validation-state: unverified
                    >  to 10.43.4.1 via ge-0/0/3.0, Push 352560
2:10.0.255.23:1::101::00:0c:29:b4:f7:5b/304 MAC/IP
                   *[EVPN/170] 00:01:49
                       Indirect
2:10.0.255.23:1::102::00:0c:29:b4:f7:5b/304 MAC/IP
                   *[EVPN/170] 00:01:41
                       Indirect
2:10.0.255.21:1::101::00:0c:29:b4:f7:51::10.1.101.1/304 MAC/IP
                   *[BGP/170] 00:43:16, localpref 100, from 10.0.255.21
                      AS path: 65101 I, validation-state: unverified
                    >  to 10.43.4.1 via ge-0/0/3.0, Push 352560
2:10.0.255.21:1::102::00:0c:29:b4:f7:51::10.1.102.1/304 MAC/IP
                   *[BGP/170] 00:33:54, localpref 100, from 10.0.255.21
                      AS path: 65101 I, validation-state: unverified
                    >  to 10.43.4.1 via ge-0/0/3.0, Push 352560
2:10.0.255.23:1::101::00:0c:29:b4:f7:5b::10.1.101.2/304 MAC/IP
                   *[EVPN/170] 00:01:49
                       Indirect
2:10.0.255.23:1::102::00:0c:29:b4:f7:5b::10.1.102.2/304 MAC/IP
                   *[EVPN/170] 00:01:41
                       Indirect
3:10.0.255.21:1::101::10.0.255.21/248 IM
                   *[BGP/170] 3d 19:06:24, localpref 100, from 10.0.255.21
                      AS path: 65101 I, validation-state: unverified
                    >  to 10.43.4.1 via ge-0/0/3.0, Push 352560
3:10.0.255.21:1::102::10.0.255.21/248 IM
                   *[BGP/170] 3d 19:06:24, localpref 100, from 10.0.255.21
```

```
                      AS path: 65101 I, validation-state: unverified
                    >  to 10.43.4.1 via ge-0/0/3.0, Push 352560
3:10.0.255.23:1::101::10.0.255.23/248 IM
                   *[EVPN/170] 3d 19:06:20
                       Indirect
3:10.0.255.23:1::102::10.0.255.23/248 IM
                   *[EVPN/170] 3d 19:06:20
                       Indirect
```

The final verification step is to ensure you can pass traffic between C1-1 and C1-2:

```
lab@vSRX-VR> ping 10.1.101.2 routing-instance CUST1-1
PING 10.1.101.2 (10.1.101.2): 56 data bytes
64 bytes from 10.1.101.2: icmp_seq=0 ttl=64 time=40.619 ms
64 bytes from 10.1.101.2: icmp_seq=1 ttl=64 time=11.686 ms
64 bytes from 10.1.101.2: icmp_seq=2 ttl=64 time=6.083 ms
64 bytes from 10.1.101.2: icmp_seq=3 ttl=64 time=8.544 ms
^C
--- 10.1.101.2 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max/stddev = 6.083/16.733/40.619/13.933 ms


lab@vSRX-VR> ping 10.1.102.2 routing-instance CUST1-1
PING 10.1.102.2 (10.1.102.2): 56 data bytes
64 bytes from 10.1.102.2: icmp_seq=0 ttl=64 time=24.610 ms
64 bytes from 10.1.102.2: icmp_seq=1 ttl=64 time=14.183 ms
64 bytes from 10.1.102.2: icmp_seq=2 ttl=64 time=81.739 ms
64 bytes from 10.1.102.2: icmp_seq=3 ttl=64 time=137.729 ms
^C
--- 10.1.102.2 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max/stddev = 14.183/64.565/137.729/49.454 m
```

Success! Traffic is successfully flowing between C1-1 and C1-2 on both VLANs.

Discussion

This recipe looked at the various steps required to enable support of EVPN over MPLS and how to configure an EVPN VLAN-aware bundle service. An EVPN VLAN-aware bundle service is a great solution when VLAN normalization is a requirement or there are overlapping MAC addresses within the EVI. It also serves as a good solution for keeping traffic separated between VLANs through dedicated bridge domains. Note that an EVPN VLAN-aware bundle carries some additional overhead, so keep that in mind when taking it under consideration.

Recipe 10: Configuring EVPN VLAN Bundle Service on Juniper MX

By Dan Hearty

- Junos OS Used: 18.4R1.8
- Juniper Platforms General Applicability: MX Series Routers

Ethernet VPN (EVPN) is technology that was initially developed to overcome some of the inherent limitations of VPLS. Such limitations include multihoming and redundancy (single-active), inefficient flooding for MAC learning, and a lack of Layer 3 support. EVPN helps solve these problems and serves as a great solution for interconnecting Layer 2 domains, among many other benefits.

In the control plane, EVPN leverages BGP via a dedicated network layer reachability information (NLRI). In the data plane, a number of encapsulation types are supported, however VXLAN and MPLS remain the most popular options. More specifically, VXLAN is a popular choice within the data center and serves as a good solution for Data Center Interconnect (DCI). MPLS, on the other hand, remains very popular in the service provider world.

EVPN currently offers three primary service types: VLAN based, VLAN bundle, and VLAN-aware bundle. Each service type provides various benefits. This recipe focuses specifically on *VLAN bundle service,* which enables multiple VLANs to map to a single bridge domain within a single EVPN instance. As a result, the EVPN VLAN bundle consumes just a single VRF target and a single MPLS label for all VLANs within the EVI, as shown in Figure 10.1. This helps keep the number of routes and labels consumed within the core network to a minimum, thus reducing control plane load.

EVPN VLAN bundle is typically not a widely deployed service type given its inherent limitations. Sharing a single bridge domain between all VLANs and maintaining unique MACs can prove challenging. However, if there's a requirement to keep control plane consumption to a minimum, with regard to routes and labels, then EVPN VLAN bundle may be a viable solution.

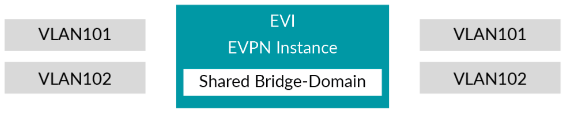


Figure 10.1 *VLAN Bundle*

Problem

In this scenario, there are two customer sites, C1-1 and C1-2, that are connected via an MPLS enabled core network. Multiple VLANs are in use at each site and there is a requirement to provide Layer 2 connectivity between the sites. EVPN must be used, however, due to control plane restrictions, only a single EVI is allowed. Furthermore, only a single label is allowed. MAC addresses across all VLANs are unique and there is no requirement for VLAN normalization.

Solution

To solve this problem, an EVPN VLAN bundle service will be configured on the MX devices to provide connectivity between the customer sites. EVPN VLAN bundle service allows multiple VLANs to be mapped to a single bridge domain within a single EVI. This results in all VLANs bundled within the EVI, sharing a single MPLS label for the service. Thus, the number of MPLS labels and EVPN routes are kept to a minimum. As all MAC addresses are unique and there is no requirement for VLAN normalization, a VLAN bundle service serves as a good option for solving this problem.

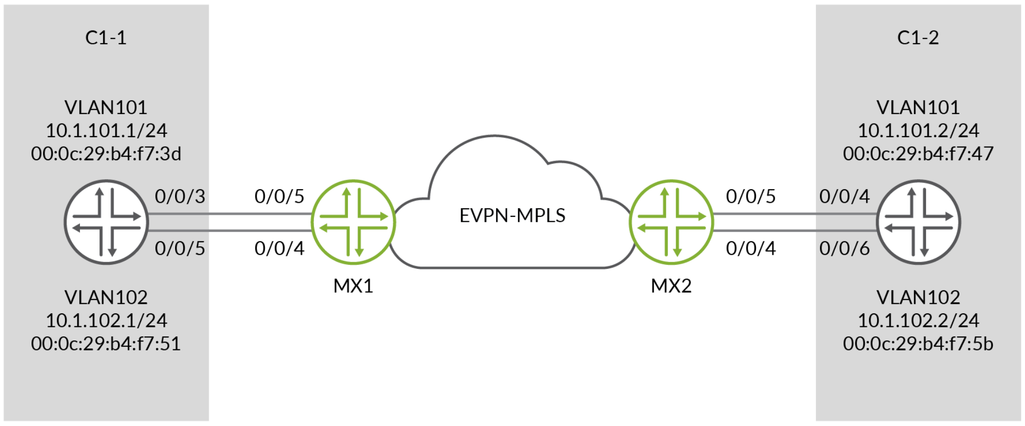


Figure 10.2 *Topology for Recipe 10's VLAN Bundle*

Overview

This recipe details the steps required to configure and verify an EVPN VLAN bundle service between Juniper vMX devices, enabling the core for IGP, MPLS, and BGP.

Configure the CE

In this example, a single vSRX is used to simulate customer site C1-1 and C1-2. Virtual routers are used to separate control plane and data plane functions between each site. For more information on how to configure routing instances, refer to this book's Recipe #1: *Virtualizing Routers with Routing Instances* by Martin Brown.

Configure CE Interfaces

We'll kick things off by configuring the interfaces that are used by C1-1 and C1-2 routing instances. Each interface is configured as an 802.1Q tagged subinterface with `vlan-tagging` enabled. Each interface is assigned an IP address within its associated VLANs. We'll use these later for verification.

NOTE When using the EVPN VLAN bundle service, it's important to ensure that MAC addresses are unique within the EVI. This is due to the shared bridge domain used by all VLANs within the instance:

```
lab@vSRX-VR> show configuration interfaces
ge-0/0/3 {
    vlan-tagging;
    unit 102 {
        vlan-id 102;
        family inet {
            address 10.1.102.1/24;
        }
    }
}
ge-0/0/4 {
    vlan-tagging;
    unit 102 {
        vlan-id 102;
        family inet {
            address 10.1.102.2/24;
        }
    }
}
ge-0/0/5 {
    vlan-tagging;
    unit 101 {
        vlan-id 101;
        family inet {
            address 10.1.101.1/24;
        }
    }
}
```

```
ge-0/0/6 {
    vlan-tagging;
    unit 101 {
        vlan-id 101;
        family inet {
            address 10.1.101.2/24;
        }
    }
}

lab@vSRX-VR> show configuration interfaces | display set
set interfaces ge-0/0/3 vlan-tagging
set interfaces ge-0/0/3 unit 102 vlan-id 102
set interfaces ge-0/0/3 unit 102 family inet address 10.1.102.1/24
set interfaces ge-0/0/4 vlan-tagging
set interfaces ge-0/0/4 unit 102 vlan-id 102
set interfaces ge-0/0/4 unit 102 family inet address 10.1.102.2/24
set interfaces ge-0/0/5 vlan-tagging
set interfaces ge-0/0/5 unit 101 vlan-id 101
set interfaces ge-0/0/5 unit 101 family inet address 10.1.101.1/24
set interfaces ge-0/0/6 vlan-tagging
set interfaces ge-0/0/6 unit 101 vlan-id 101
set interfaces ge-0/0/6 unit 101 family inet address 10.1.101.2/24
```

Configure C1-1 and C1-2 Routing Instances

Here we create two `virtual-router routing-instances` named C1-1 & C1-2 and assign the interfaces, configured in the previous step, to the relevant `routing-instance`:

```
lab@vSRX-VR> show configuration routing-instances
C1-1 {
    instance-type virtual-router;
    interface ge-0/0/3.102;
    interface ge-0/0/5.101;
}
C1-2 {
    instance-type virtual-router;
    interface ge-0/0/4.102;
    interface ge-0/0/6.101;

lab@vSRX-VR> show configuration routing-instances | display set
set routing-instances C1-1 instance-type virtual-router
set routing-instances C1-1 interface ge-0/0/3.102
set routing-instances C1-1 interface ge-0/0/5.101
set routing-instances C1-2 instance-type virtual-router
set routing-instances C1-2 interface ge-0/0/4.102
set routing-instances C1-2 interface ge-0/0/6.101
```

Configure MX1

The following section details all the necessary steps required to configure IGP, MPLS, BGP, and an EVPN VLAN bundle service on MX1. You may want to skip to the last section *Configure VLAN Bundle Service* if you already have an MPLS core network enabled.

Configure IGP

ISIS is used in the IGP to provide loopback connectivity between MX1 and MX2.

NOTE Ensure family ISO is enabled on core facing interfaces and also ensure a valid ISO address is configured on lo0:

```
lab@vMX1> show configuration protocols
isis {
    interface ge-0/0/0.11 {
        point-to-point;
    }
    interface lo0.0;
}

lab@vMX1> show configuration protocols isis | display set
set protocols isis interface ge-0/0/0.11 point-to-point
set protocols isis interface lo0.0
```

Configure MPLS

MPLS is enabled on core facing interfaces.

NOTE Ensure family MPLS is enabled on core facing interfaces:

```
lab@vMX1> show configuration protocols
mpls {
    interface ge-0/0/0.11;

lab@vMX1> show configuration protocols mpls | display set
set protocols mpls interface ge-0/0/0.11
```

Configure LDP

LDP is configured for lo0 and the core facing interface:

```
lab@vMX1> show configuration protocols
ldp {
    interface ge-0/0/0.11;
    interface lo0.0;

lab@vMX1> show configuration protocols ldp | display set
set protocols ldp interface ge-0/0/0.11
set protocols ldp interface lo0.0
```

Configure BGP

Now that we've enabled loopback reachability between MX1 and MX2, and LDP has been enabled, BGP can be configured. A group named EVPN is created with an EBGP peering to MX2. The `multihop` knob is required as the TTL is greater than 1 to reach MX2. Note that `family evpn signaling` is configured for the peering. This enables the exchange of EVPN information only and overrides the default family inet setting:

```
lab@vMX1> **show configuration protocols**
bgp {
    group EVPN {
        type external;
        multihop;
        local-address 10.0.255.21;
        family evpn {
            signaling;
        }
        local-as 65101;
        neighbor 10.0.255.23 {
            peer-as 65001;
        }
    }
}

lab@vMX1> **show configuration protocols bgp | display set**
set protocols bgp group EVPN type external
set protocols bgp group EVPN multihop
set protocols bgp group EVPN local-address 10.0.255.21
set protocols bgp group EVPN family evpn signaling
set protocols bgp group EVPN local-as 65101
set protocols bgp group EVPN neighbor 10.0.255.23 peer-as 65001
```

Configure CE-Facing Interface

An EVPN VLAN bundle does not support the use of trunk interfaces, thus we must configure 802.1Q tagged sub-interfaces. The PE-CE interface is configured to accept VLAN tagged frames from C1-1 VLAN101 and VLAN102. `Encapsulation vlan-bridge` along with `family bridge` is required:

```
lab@vMX1> **show configuration interfaces**
ge-0/0/4 {
    flexible-vlan-tagging;
    encapsulation flexible-ethernet-services;
    unit 101 {
        encapsulation vlan-bridge;
        vlan-id 101;
        family bridge;
    }
}
ge-0/0/5 {
    flexible-vlan-tagging;
    encapsulation flexible-ethernet-services;
    unit 102 {
        encapsulation vlan-bridge;
        vlan-id 102;
        family bridge;
    }
}


lab@vMX1> **show configuration interfaces | display set**
set interfaces ge-0/0/4 flexible-vlan-tagging
set interfaces ge-0/0/4 encapsulation flexible-ethernet-services
set interfaces ge-0/0/4 unit 101 encapsulation vlan-bridge
set interfaces ge-0/0/4 unit 101 vlan-id 101
set interfaces ge-0/0/4 unit 101 family bridge
set interfaces ge-0/0/5 flexible-vlan-tagging
set interfaces ge-0/0/5 encapsulation flexible-ethernet-services
set interfaces ge-0/0/5 unit 102 encapsulation vlan-bridge
set interfaces ge-0/0/5 unit 102 vlan-id 102
set interfaces ge-0/0/5 unit 102 family bridge
```

Configure VLAN-Bundle Service

Configuring an EVPN VLAN bundle involves creating an EVPN routing instance, assigning PE-CE interfaces, and enabling EVPN. Also, as we are using MPLS for transport, there must be a `route-distinguisher` and `vrf-target` assigned for the service.

NOTE EVPN VLAN bundle creates a single shared bridge domain for the EVPN instance. Thus, all broadcast domains share the same bridge table. This is why there is no requirement to configure a `bridge-domain` within the EVPN instance:

```
lab@vMX1> show configuration routing-instances
CUST1 {
    instance-type evpn;
    vlan-id none;
    interface ge-0/0/4.101;
    interface ge-0/0/5.102;
    route-distinguisher 10.0.255.21:1;
    vrf-target target:65000:100;
    protocols {
        evpn;
    }
}

lab@vMX1> show configuration routing-instances | display set
set routing-instances CUST1 instance-type evpn
set routing-instances CUST1 vlan-id none
set routing-instances CUST1 interface ge-0/0/4.101
set routing-instances CUST1 interface ge-0/0/5.102
set routing-instances CUST1 route-distinguisher 10.0.255.21:1
set routing-instances CUST1 vrf-target target:65000:100
set routing-instances CUST1 protocols evpn
```

Configure MX2

The following section is very similar to the previous section, although this time we focus on MX2. Once again, you may want to skip to the last section *Configure VLAN Bundle Service* if you already have an MPLS core network enabled.

Configure IGP

ISIS is used in the IGP to provide loopback connectivity between MX2 and MX1.

NOTE Ensure family ISO is enabled on core facing interfaces and also ensure a valid ISO address is configured on lo0:

```
lab@vMX2> show configuration protocols
isis {
    interface ge-0/0/3.0 {
        point-to-point;
    }
    interface lo0.0;
}

lab@vMX2> show configuration protocols isis | display set
set protocols isis interface ge-0/0/3.0 point-to-point
set protocols isis interface lo0.0
```

Configure MPLS

MPLS is enabled on core-facing interfaces.

NOTE Ensure family MPLS is enabled on core-facing interfaces:

```
lab@vMX2> show configuration protocols
mpls {
    interface ge-0/0/3.0;
}

lab@vMX2> show configuration protocols mpls | display set
set protocols mpls interface ge-0/0/3.0
```

Configure LDP

LDP is configured for lo0 and the core-facing interface:

```
lab@vMX2> show configuration protocols
ldp {
    interface ge-0/0/3.0;
    interface lo0.0;
}

lab@vMX2> show configuration protocols ldp | display set
set protocols ldp interface ge-0/0/3.0
set protocols ldp interface lo0.0
```

Configure BGP

Now that we've enabled loopback reachability between MX2 and MX1, and LDP has been enabled, BGP can be configured. A group named `EVPN` is created with an EBGP peering to MX1. The `multihop` knob is required as the TTL is greater than 1 to reach MX1. Note that `family evpn signaling` is configured for the peering. This enables the exchange of EVPN information only and overrides the default `family inet` setting:

```
lab@vMX2> show configuration protocols
bgp {
    group EVPN {
        type external;
        multihop;
        local-address 10.0.255.23;
        family evpn {
            signaling;
        }
        local-as 65001;
        neighbor 10.0.255.21 {
            peer-as 65101;
        }
    }
}

lab@vMX2> show configuration protocols bgp | display set
set protocols bgp group EVPN type external
set protocols bgp group EVPN multihop
set protocols bgp group EVPN local-address 10.0.255.23
set protocols bgp group EVPN family evpn signaling
set protocols bgp group EVPN local-as 65001
set protocols bgp group EVPN neighbor 10.0.255.21 peer-as 65101
```

Configure CE-Facing Interface

PE-CE interfaces are configured to accept VLAN tagged frames from the vSRX VR device. `Encapsulation vlan-bridge` along with `family bridge` is required:

```
lab@vMX1> show configuration interfaces
ge-0/0/4 {
    flexible-vlan-tagging;
    encapsulation flexible-ethernet-services;
    unit 101 {
        encapsulation vlan-bridge;
        vlan-id 101;
        family bridge;
    }
}
ge-0/0/5 {
    flexible-vlan-tagging;
    encapsulation flexible-ethernet-services;
    unit 102 {
        encapsulation vlan-bridge;
        vlan-id 102;
        family bridge;
    }
}


lab@vMX1> show configuration interfaces | display set
set interfaces ge-0/0/4 flexible-vlan-tagging
set interfaces ge-0/0/4 encapsulation flexible-ethernet-services
set interfaces ge-0/0/4 unit 101 encapsulation vlan-bridge
set interfaces ge-0/0/4 unit 101 vlan-id 101
set interfaces ge-0/0/4 unit 101 family bridge
set interfaces ge-0/0/5 flexible-vlan-tagging
set interfaces ge-0/0/5 encapsulation flexible-ethernet-services
set interfaces ge-0/0/5 unit 102 encapsulation vlan-bridge
set interfaces ge-0/0/5 unit 102 vlan-id 102
set interfaces ge-0/0/5 unit 102 family bridge
```

Configure VLAN-Bundle Service

Configuring an EVPN VLAN bundle involves creating an EVPN routing instance, assigning PE-CE interfaces and enabling EVPN. Also, as we are using MPLS for transport, there must be a `route-distinguisher` and `vrf-target` assigned for the service.

NOTE An EVPN VLAN bundle creates a single shared bridge domain for the EVPN instance. Thus, all broadcast domains share the same bridge table. This is why there is no requirement to configure a `bridge-domain` within the EVPN instance:

```
lab@vMX2> show configuration routing-instances
CUST1 {
    instance-type evpn;
    vlan-id none;
    interface ge-0/0/4.101;
    interface ge-0/0/5.102;
    route-distinguisher 10.0.255.23:1;
    vrf-target target:65000:100;
    protocols {
        evpn;
    }
}

lab@vMX2> show configuration routing-instances | display set
set routing-instances CUST1 instance-type evpn
set routing-instances CUST1 vlan-id none
set routing-instances CUST1 interface ge-0/0/4.101
set routing-instances CUST1 interface ge-0/0/5.102
set routing-instances CUST1 route-distinguisher 10.0.255.23:1
set routing-instances CUST1 vrf-target target:65000:100
set routing-instances CUST1 protocols evpn
```

Verification

The following section details the steps required to verify IGP, LDP, BGP, and the EVPN VLAN bundle service.

IGP, LDP

First, we'll verify that ISIS and LDP has been established with the core network. We'll also ensure there is an MPLS label to reach the loopback of the remote MX device:

MX1

```
lab@vMX1> show isis adjacency
Interface             System         L State        Hold (secs) SNPA
ge-0/0/0.11           vSRX1          3  Up                   20


lab@vMX1> show ldp neighbor
Address                            Interface       Label space ID     Hold time
10.41.11.1                         ge-0/0/0.11     10.0.255.11:0        12
```

```
lab@vMX1> show route 10.0.255.23/32

inet.0: 33 destinations, 33 routes (33 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both


10.0.255.23/32     *[IS-IS/15] 1w5d 10:10:46, metric 30
                    >  to 10.41.11.1 via ge-0/0/0.11

inet.3: 6 destinations, 6 routes (6 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

10.0.255.23/32     *[LDP/9] 1w5d 10:10:45, metric 1
                    >  to 10.41.11.1 via ge-0/0/0.11, Push 346272
```

MX2

```
lab@vMX2> show isis adjacency
Interface             System         L State        Hold (secs) SNPA
ge-0/0/3.0            vSRX3          3  Up                   24


lab@vMX2> show ldp neighbor
Address                            Interface       Label space ID     Hold time
10.43.4.1                          ge-0/0/3.0      10.0.255.13:0        13


lab@vMX2> show route 10.0.255.21/32

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

10.0.255.21/32     *[IS-IS/15] 1w5d 10:14:54, metric 30
                    >  to 10.43.4.1 via ge-0/0/3.0

inet.3: 6 destinations, 6 routes (6 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

10.0.255.21/32     *[LDP/9] 1w5d 10:15:04, metric 1
                    >  to 10.43.4.1 via ge-0/0/3.0, Push 352560
```

The above output verifies that ISIS and LDP have been established with the core network. You can also see that a /32 route has been installed in inet.0 and inet.3 for the remote MX loopback. On MX1, LDP has assigned the label value 346272 to reach MX2. On MX2, LDP has assigned the label value 352560 to reach MX1.

BGP

Now that we have an entry in inet.0 for the remote loopback, and BGP can resolve via inet.3, we can verify BGP.

In this scenario we're using BGP to carry EVPN information. This was enabled by the `family evpn signaling` knob under the BGP configuration group:

MX1

```
lab@vMX1> show bgp neighbor 10.0.255.23
Peer: 10.0.255.23+179 AS 65001 Local: 10.0.255.21+65167 AS 65101
  Group: EVPN                  Routing-Instance: master
  Forwarding routing-instance: master
  Type: External    State: Established    Flags: <Sync>
  Last State: OpenConfirm   Last Event: RecvKeepAlive
  Last Error: None
  Options: <Multihop Preference LocalAddress AddressFamily PeerAS LocalAS Rib-group Refresh>
  Address families configured: evpn
  Local Address: 10.0.255.21 Holdtime: 90 Preference: 170 Local AS: 65101 Local System AS: 65000
  Number of flaps: 0
  Peer ID: 10.0.255.23     Local ID: 10.0.255.21       Active Holdtime: 90
  Keepalive Interval: 30         Group index: 0    Peer index: 0    SNMP index: 0
  I/O Session Thread: bgpio-0 State: Enabled
  BFD: disabled, down
  NLRI for restart configured on peer: evpn
  NLRI advertised by peer: evpn
  NLRI for this session: evpn
  Peer supports Refresh capability (2)
  Stale routes from peer are kept for: 300
  Peer does not support Restarter functionality
  Restart flag received from the peer: Notification
  NLRI that restart is negotiated for: evpn
  NLRI of received end-of-rib markers: evpn
  NLRI of all end-of-rib markers sent: evpn
  Peer does not support LLGR Restarter functionality
  Peer supports 4 byte AS extension (peer-as 65001)
  Peer does not support Addpath
  Table CUST1.evpn.0
    RIB State: BGP restart is complete
    RIB State: VPN restart is complete
    Send state: not advertising
    Active prefixes:              5
    Received prefixes:            5
    Accepted prefixes:            5
    Suppressed due to damping:    0
  Table __default_evpn__.evpn.0
    RIB State: BGP restart is complete
    RIB State: VPN restart is complete
    Send state: not advertising
    Active prefixes:              0
    Received prefixes:            0
    Accepted prefixes:            0
    Suppressed due to damping:    0
  Table bgp.evpn.0 Bit: 20000
    RIB State: BGP restart is complete
    RIB State: VPN restart is complete
    Send state: in sync
    Active prefixes:              5
    Received prefixes:            5
    Accepted prefixes:            5
    Suppressed due to damping:    0
    Advertised prefixes:          5
  Last traffic (seconds): Received 10   Sent 6    Checked 1074958
  Input messages:  Total 41588  Updates 1926    Refreshes 0     Octets 984017
  Output messages: Total 41521  Updates 1704    Refreshes 0     Octets 961177
  Output Queue[1]: 0            (bgp.evpn.0, evpn)
```

MX2

```
lab@vMX2> show bgp neighbor 10.0.255.21
Peer: 10.0.255.21+65167 AS 65101 Local: 10.0.255.23+179 AS 65001
  Group: EVPN                  Routing-Instance: master
  Forwarding routing-instance: master
  Type: External    State: Established    Flags: <Sync>
  Last State: OpenConfirm   Last Event: RecvKeepAlive
  Last Error: None
  Options: <Multihop Preference LocalAddress AddressFamily PeerAS LocalAS Rib-group Refresh>
  Address families configured: evpn
  Local Address: 10.0.255.23 Holdtime: 90 Preference: 170 Local AS: 65001 Local System AS: 65000
  Number of flaps: 0
  Peer ID: 10.0.255.21     Local ID: 10.0.255.23       Active Holdtime: 90
  Keepalive Interval: 30         Group index: 1    Peer index: 0    SNMP index: 1
  I/O Session Thread: bgpio-0 State: Enabled
  BFD: disabled, down
  NLRI for restart configured on peer: evpn
  NLRI advertised by peer: evpn
  NLRI for this session: evpn
  Peer supports Refresh capability (2)
  Stale routes from peer are kept for: 300
  Peer does not support Restarter functionality
  Restart flag received from the peer: Notification
  NLRI that restart is negotiated for: evpn
  NLRI of received end-of-rib markers: evpn
  NLRI of all end-of-rib markers sent: evpn
  Peer does not support LLGR Restarter functionality
  Peer supports 4 byte AS extension (peer-as 65101)
  Peer does not support Addpath
  Table CUST1.evpn.0
    RIB State: BGP restart is complete
    RIB State: VPN restart is complete
    Send state: not advertising
    Active prefixes:             5
    Received prefixes:           5
    Accepted prefixes:           5
    Suppressed due to damping:   0
  Table __default_evpn__.evpn.0
    RIB State: BGP restart is complete
    RIB State: VPN restart is complete
    Send state: not advertising
    Active prefixes:             0
    Received prefixes:           0
    Accepted prefixes:           0
    Suppressed due to damping:   0
  Table bgp.evpn.0 Bit: 30000
    RIB State: BGP restart is complete
    RIB State: VPN restart is complete
    Send state: in sync
    Active prefixes:             5
    Received prefixes:           5
    Accepted prefixes:           5
    Suppressed due to damping:   0
    Advertised prefixes:         5
  Last traffic (seconds): Received 9    Sent 16   Checked 1074991
  Input messages:  Total 41524  Updates 1705    Refreshes 0     Octets 961278
  Output messages: Total 41589  Updates 1925    Refreshes 0     Octets 984036
  Output Queue[2]: 0            (bgp.evpn.0, evpn)
```

Confirmation that BGP has been successfully established between MX1 and MX3 and the session is enabled to carry EVPN information.

EVPN VLAN Bundle Service

The final step verifies that customer MAC addresses are imported into EVPN and exchanged between the MX devices.

The EVPN database provides information about locally-learned MAC addresses and includes the interface on which the MAC was received. It also displays remotely-learned MACs and includes the source, therefore the remote PE:

MX1

```
lab@vMX1> show evpn database
Instance: CUST1
VLAN  DomainId  MAC address        Active source                  Timestamp        IP address
                00:0c:29:b4:f7:3d  ge-0/0/5.102                   Feb 21 13:58:06  10.1.102.1
                00:0c:29:b4:f7:47  10.0.255.23                    Feb 21 13:49:04  10.1.102.2
                00:0c:29:b4:f7:51  ge-0/0/4.101                   Feb 21 14:15:24  10.1.101.1
                00:0c:29:b4:f7:5b  10.0.255.23                    Feb 21 13:28:06  10.1.101.2
```

MX2

```
lab@vMX2> show evpn database
Instance: CUST1
VLAN  DomainId  MAC address        Active source                  Timestamp        IP address
                00:0c:29:b4:f7:3d  10.0.255.21                    Feb 21 13:45:01  10.1.102.1
                00:0c:29:b4:f7:47  ge-0/0/5.102                   Feb 21 13:54:04  10.1.102.2
                00:0c:29:b4:f7:51  10.0.255.21                    Feb 21 13:50:59  10.1.101.1
                00:0c:29:b4:f7:5b  ge-0/0/4.101                   Feb 21 14:13:12  10.1.101.2
```

Once the locally-learned MAC addresses have been imported into the local EVPN table, BGP is used to advertise details to the remote MX. BGP RIB-OUT for the CUST1 instance shows what's being announced to the remote MX. Note that each MAC is represented by two EVPN type2 routes. There is also an EVPN type3 route for the instance. This is used to deal with BUM traffic:

MX1

```
lab@vMX1> show route advertising-protocol bgp 10.0.255.23 table CUST1.evpn.0

CUST1.evpn.0: 10 destinations, 10 routes (10 active, 0 holddown, 0 hidden)
  Prefix                  Nexthop              MED     Lclpref    AS path
  2:10.0.255.21:1::0::00:0c:29:b4:f7:3d/304 MAC/IP
*                         Self                                    I
  2:10.0.255.21:1::0::00:0c:29:b4:f7:51/304 MAC/IP
*                         Self                                    I
  2:10.0.255.21:1::0::00:0c:29:b4:f7:3d::10.1.102.1/304 MAC/IP
*                         Self                                    I
  2:10.0.255.21:1::0::00:0c:29:b4:f7:51::10.1.101.1/304 MAC/IP
*                         Self                                    I
  3:10.0.255.21:1::0::10.0.255.21/248 IM
*                         Self                                    I
```

MX2

```
lab@vMX2> show route advertising-protocol bgp 10.0.255.21 table CUST1.evpn.0

CUST1.evpn.0: 10 destinations, 10 routes (10 active, 0 holddown, 0 hidden)
  Prefix                  Nexthop              MED     Lclpref    AS path
  2:10.0.255.23:1::0::00:0c:29:b4:f7:47/304 MAC/IP
*                         Self                                    I
  2:10.0.255.23:1::0::00:0c:29:b4:f7:5b/304 MAC/IP
*                         Self                                    I
  2:10.0.255.23:1::0::00:0c:29:b4:f7:47::10.1.102.2/304 MAC/IP
*                         Self                                    I
  2:10.0.255.23:1::0::00:0c:29:b4:f7:5b::10.1.101.2/304 MAC/IP
*                         Self                                    I
  3:10.0.255.23:1::0::10.0.255.23/248 IM
*                         Self                                    I
```

Based on the `route-target` configured under the `routing-instance`, EVPN routes are imported into the relevant instance. By checking the CUST1 routing table, you can see all local and remote MACs that are participating in the EVPN VLAN bundle service:

MX1

```
lab@vMX1> show route table CUST1.evpn.0

CUST1.evpn.0: 10 destinations, 10 routes (10 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

2:10.0.255.21:1::0::00:0c:29:b4:f7:3d/304 MAC/IP
                   *[EVPN/170] 00:35:31
                       Indirect
2:10.0.255.21:1::0::00:0c:29:b4:f7:51/304 MAC/IP
                   *[EVPN/170] 00:08:12
                       Indirect
2:10.0.255.23:1::0::00:0c:29:b4:f7:47/304 MAC/IP
                   *[BGP/170] 00:44:32, localpref 100, from 10.0.255.23
                      AS path: 65001 I, validation-state: unverified
                    >  to 10.41.11.1 via ge-0/0/0.11, Push 346272
2:10.0.255.23:1::0::00:0c:29:b4:f7:5b/304 MAC/IP
                   *[BGP/170] 01:05:30, localpref 100, from 10.0.255.23
                      AS path: 65001 I, validation-state: unverified
                    >  to 10.41.11.1 via ge-0/0/0.11, Push 346272
2:10.0.255.21:1::0::00:0c:29:b4:f7:3d::10.1.102.1/304 MAC/IP
                   *[EVPN/170] 00:35:31
                       Indirect
2:10.0.255.21:1::0::00:0c:29:b4:f7:51::10.1.101.1/304 MAC/IP
                   *[EVPN/170] 00:08:12
                       Indirect
2:10.0.255.23:1::0::00:0c:29:b4:f7:47::10.1.102.2/304 MAC/IP
                   *[BGP/170] 00:44:32, localpref 100, from 10.0.255.23
                      AS path: 65001 I, validation-state: unverified
                    >  to 10.41.11.1 via ge-0/0/0.11, Push 346272
2:10.0.255.23:1::0::00:0c:29:b4:f7:5b::10.1.101.2/304 MAC/IP
                   *[BGP/170] 01:05:30, localpref 100, from 10.0.255.23
                      AS path: 65001 I, validation-state: unverified
                    >  to 10.41.11.1 via ge-0/0/0.11, Push 346272
3:10.0.255.21:1::0::10.0.255.21/248 IM
```

```
                   *[EVPN/170] 1w5d 13:10:31
                       Indirect
3:10.0.255.23:1::0::10.0.255.23/248 IM
                   *[BGP/170] 1w5d 13:09:25, localpref 100, from 10.0.255.23
                      AS path: 65001 I, validation-state: unverified
                    >  to 10.41.11.1 via ge-0/0/0.11, Push 346272
```

MX2

```
lab@vMX2> show route table CUST1.evpn.0

CUST1.evpn.0: 10 destinations, 10 routes (10 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

2:10.0.255.21:1::0::00:0c:29:b4:f7:3d/304 MAC/IP
                   *[BGP/170] 00:50:37, localpref 100, from 10.0.255.21
                      AS path: 65101 I, validation-state: unverified
                    >  to 10.43.4.1 via ge-0/0/3.0, Push 352560
2:10.0.255.21:1::0::00:0c:29:b4:f7:51/304 MAC/IP
                   *[BGP/170] 00:44:39, localpref 100, from 10.0.255.21
                      AS path: 65101 I, validation-state: unverified
                    >  to 10.43.4.1 via ge-0/0/3.0, Push 352560
2:10.0.255.23:1::0::00:0c:29:b4:f7:47/304 MAC/IP
                   *[EVPN/170] 00:41:33
                       Indirect
2:10.0.255.23:1::0::00:0c:29:b4:f7:5b/304 MAC/IP
                   *[EVPN/170] 00:04:35
                       Indirect
2:10.0.255.21:1::0::00:0c:29:b4:f7:3d::10.1.102.1/304 MAC/IP
                   *[BGP/170] 00:50:37, localpref 100, from 10.0.255.21
                      AS path: 65101 I, validation-state: unverified
                    >  to 10.43.4.1 via ge-0/0/3.0, Push 352560
2:10.0.255.21:1::0::00:0c:29:b4:f7:51::10.1.101.1/304 MAC/IP
                   *[BGP/170] 00:44:39, localpref 100, from 10.0.255.21
                      AS path: 65101 I, validation-state: unverified
                    >  to 10.43.4.1 via ge-0/0/3.0, Push 352560
2:10.0.255.23:1::0::00:0c:29:b4:f7:47::10.1.102.2/304 MAC/IP
                   *[EVPN/170] 00:41:33
                       Indirect
2:10.0.255.23:1::0::00:0c:29:b4:f7:5b::10.1.101.2/304 MAC/IP
                   *[EVPN/170] 00:04:35
                       Indirect
3:10.0.255.21:1::0::10.0.255.21/248 IM
                   *[BGP/170] 1w5d 13:11:26, localpref 100, from 10.0.255.21
                      AS path: 65101 I, validation-state: unverified
                    >  to 10.43.4.1 via ge-0/0/3.0, Push 352560
3:10.0.255.23:1::0::10.0.255.23/248 IM
                   *[EVPN/170] 1w5d 13:11:42
                       Indirect
```

The final verification step is to ensure we can pass traffic between C1-1 and C1-2:

```
lab@vSRX-VR> ping 10.1.101.2 routing-instance CUST1-1
PING 10.1.101.2 (10.1.101.2): 56 data bytes
64 bytes from 10.1.101.2: icmp_seq=0 ttl=64 time=162.051 ms
64 bytes from 10.1.101.2: icmp_seq=1 ttl=64 time=137.678 ms
64 bytes from 10.1.101.2: icmp_seq=2 ttl=64 time=34.656 ms
64 bytes from 10.1.101.2: icmp_seq=3 ttl=64 time=4.520 ms
```

```
^C
--- 10.1.101.2 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max/stddev = 4.520/84.726/162.051/66.564 ms

lab@vSRX-VR> ping 10.1.102.2 routing-instance CUST1-1
PING 10.1.102.2 (10.1.102.2): 56 data bytes
64 bytes from 10.1.102.2: icmp_seq=0 ttl=64 time=14.031 ms
64 bytes from 10.1.102.2: icmp_seq=1 ttl=64 time=40.817 ms
64 bytes from 10.1.102.2: icmp_seq=2 ttl=64 time=22.551 ms
64 bytes from 10.1.102.2: icmp_seq=3 ttl=64 time=9.126 ms
^C
--- 10.1.102.2 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max/stddev = 9.126/21.631/40.817/12.074 ms
```

Success! Traffic is flowing between C1-1 and C1-2 on both VLANs, although it's important to note that EVPN VLAN bundle service shares a single bridge domain within the EVPN instance.

Discussion

This recipe looked at the various steps required to enable support of EVPN over MPLS by configuring an EVPN VLAN bundle service. EVPN VLAN bundle service is a great solution when VLAN normalization is not a requirement and unique MAC addresses are in use within the EVI. It also serves as a good solution when control plane utilization is a concern and keeps MPLS label consumption to a minimum. Note that a single bridge domain is used for the entire EVI, thus there is no separation between VLANs in the domain.

Recipe 11: Using Terminating Actions in Junos Routing Policy

By Chris Parker

- vSRX Version Used: 12.1X47-D15.4
- Juniper Platforms General Applicability: MX, EX, SRX, QFX Series

In Recipe #17 in this cookbook, Pierre-Yves Maunier shows us how to simplify our Junos BGP routing policies. As a warm-up to that recipe, let's learn what the different actions in a policy actually do, because there's more to the words "accept" and "reject" than meets the eye – And, just like a careful bribe to an international diplomat, when you use them right you're rewarded with tremendous power.

Problem

You want to manipulate a prefix in multiple ways, but for some reason, only the first term in your policy is being actioned. You want to understand how to use the `then` statement properly, so that your policy works correctly.

Solution

You may already be familiar with the two main actions in a policy: `accept` and `reject`. They do exactly what they say they will on the tin. However, one thing that Junos newbies might not intuitively know is that accept and reject are more than just actions: they're *terminating actions*. If a term in a policy has an action of accept or reject, then if a prefix matches the term, no further checks are done.

Sometimes, this is exactly what you want. For example, in Figure 11.1 there are two routers, 1 and 2, with eBGP peering.

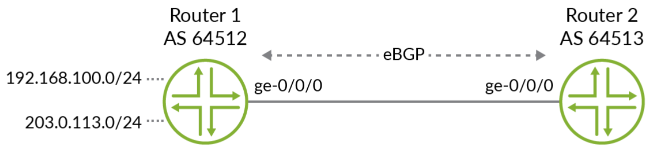


Figure 11.1 Recipe 11's Topology

Router 1 is sending two prefixes to Router 2: 192.168.100.0/24, and 203.0.113.0/24. Now, imagine you make a policy on Router 2 containing two terms: one rejecting private IPs, and one allowing everything else:

```
set policy-options policy-statement REJECT-EVIL-PREFIXES term REJECT-RFC1918 from route-
filter 10.0.0.0/8 orlonger
set policy-options policy-statement REJECT-EVIL-PREFIXES term REJECT-RFC1918 from route-
filter 172.16.0.0/12 orlonger
set policy-options policy-statement REJECT-EVIL-PREFIXES term REJECT-RFC1918 from route-
filter 192.168.0.0/16 orlonger
set policy-options policy-statement REJECT-EVIL-PREFIXES term REJECT-RFC1918 then reject
set policy-options policy-statement REJECT-EVIL-PREFIXES term ACCEPT-ALL-OTHERS then accept
```

If you apply this as an import policy on Router 2, you'll see that Router 2 only accepts the 203.0.113.0/24 prefix. Great! The private range is rejected:

```
root@Router2> show route protocol bgp

inet.0: 4 destinations, 4 routes (3 active, 0 holddown, 1 hidden)
+ = Active Route, - = Last Active, * = Both

203.0.113.0/24     *[BGP/170] 00:00:23, localpref 100
                      AS path: 64512 I
                    > to 10.10.12.1 via ge-0/0/0.0

root@Router2> show route protocol bgp hidden

inet.0: 4 destinations, 4 routes (3 active, 0 holddown, 1 hidden)
+ = Active Route, - = Last Active, * = Both

192.168.100.0/24    [BGP ] 00:00:25, localpref 100
                      AS path: 64512 I
                    > to 10.10.12.1 via ge-0/0/0.0
```

NOTE Not many people know that 203.0.113.0/24 is a range reserved for use in documentation. How handy it is to have something that looks like a public address that we can freely use! Wouldn't it be handy if people used this in VPN and NAT documentation, instead of using private IPs on both the WAN and LAN? Yes. Yes, it would.

Now, in this scenario, if our peer advertises 192.168.100.0/24 and our first statement rejects it, we don't need to check any further down the policy. We know we want to reject it, so the policy checking can safely stop there.

Sometimes, though, you do want to continue checking. For example, you might want to manipulate a prefix in some way but then carry on checking for other conditions to manipulate it further.

Let's imagine that we wanted to achieve the following: if a prefix comes from Customer A's VRF, then add community 64512:111 to the prefix and carry on checking. If the prefix is one that we want to be able to access from our management platforms (for example, the WAN IP address of the router), add community 64512:222 to the prefix and carry on checking. Finally, accept everything.

Well, we've got news for you: you don't need to imagine it. Take a look at the configuration below and see if you can spot why this particular configuration *won't* work:

```
set policy-options community target-CUSTOMER-A-COMMUNITY members target:64512:111
set policy-options community target-MPLS-MANAGEMENT-COMMUNITY members target:64512:222

set policy-options policy-statement CUSTOMER-A-EXPORT term CUSTOMER-LANS then community add target-
CUSTOMER-A-COMMUNITY
set policy-options policy-statement CUSTOMER-A-EXPORT term CUSTOMER-LANS then accept
set policy-options policy-statement CUSTOMER-A-EXPORT term MANAGEMENT from prefix-list-filter MPLS-
MANAGEMENT-PREFIXES orlonger
set policy-options policy-statement CUSTOMER-A-EXPORT term MANAGEMENT then community add target-MPLS-
MANAGEMENT-COMMUNITY
set policy-options policy-statement CUSTOMER-A-EXPORT term MANAGEMENT then accept
set policy-options policy-statement CUSTOMER-A-EXPORT term ACCEPT-ALL then accept
```

The first term in this policy checks whether the prefix came from the customer's VRF. If it did, the policy adds community 64512:111 to the prefix. However, because the action in this first term was `accept`, Junos doesn't carry on checking whether the prefix also needs to be assigned the `MANAGEMENT` community.

The result of this is that any prefix coming out of the customer's VRF will be accepted by the first term and advertised to the PE router's peers. But, if the prefix happened to have also been within our management range, our router will have advertised the prefix without the `MANAGEMENT` community, because the second term was never checked. As such, the prefix would never be imported into our management VRF around the rest of the network. It's just about the saddest story you've ever heard!

Dry your eyes, though, because we're going to fix it. If we don't explicitly add an action, the Junos router actually carries on checking the next policy! If you don't explicitly state whether to accept or reject a prefix, a routing policy has one of three default behaviors: `next-term` if there are more terms to check; `next-policy` if you're at the final term in a policy; or, if there's no policies left to check, each routing protocol has its own built-in default behavior. We'll talk about that in a moment.

This default behavior is similar to, but not the same as, `accept`: what the default behavior says is *manipulate the route characteristics like the policy term states, then carry on checking*. The prefix hasn't strictly been accepted yet; rather, it's been manipulated as necessary, and then passed on for further checking.

Even though this is the default behavior, some people like to configure these actions explicitly, so that customers and junior engineers can read through configurations more easily. Junos configurations can be a bit longer sometimes compared to other vendors, but in this author's opinion this extra verbosity makes configurations extremely readable and leads to an exponential increase in how simple it is to understand what's going on, even for people who aren't familiar with Junos. Explicitly configuring the defaults almost always helps make things even clearer still.

In that last example we didn't want the policy checks to stop, so using an action of `accept` was the wrong choice. Let's fix it by deleting the first two `accept` statements, which means that our Juniper routers will use the default action of `next-term`:

```
delete policy-options policy-statement CUSTOMER-A-EXPORT term CUSTOMER-LANS then accept
delete policy-options policy-statement CUSTOMER-A-EXPORT term MANAGEMENT then accept
```

We've left the final accept in place – it's the last term, so at the moment there's no harm in it. But remember: Junos allows you to chain as many policies together as you like, so, if in the future you add a second policy after this one, you'll want to think about whether or not to keep this final accept in place!

NOTE Sometimes, using `accept` part way through a policy might indeed be the correct choice. For example, imagine if you wanted to achieve something like this: *If a prefix matches prefix-list PRIORITY-TRAFFIC, then I want to add a certain community to the prefix, and I want to allow it. However, I don't want anything else to happen to the prefix. I've already marked it as priority traffic, so even if there are more policies it could be checked against, I want the policy checks to end here*. In this particular example, a policy with an action of `accept` in the first term would be exactly what we want.

Now, you may be wondering: what's this about each protocol having its own default behavior? What happens if you get to the very last term of the very last policy and still haven't configured a terminating action? Is the prefix accepted, or rejected? Good question! Let's investigate.

TIP Before looking at some examples, remember: whenever you're thinking about Junos routing policy, imagine yourself standing inside the inet.0 or inet6.0 table. *Import* means importing prefix from the protocol, into the routing table. *Export* means pushing prefixes from the routing table, into the protocol.

TIP Are you still imagining yourself standing inside the routing table? I don't blame you. It's a beautiful place. Feel free to stay there for a few minutes if you like. Treat yourself to some quality time.

Let's start with BGP. If you don't put a policy on a BGP peering, here's what happens:

IMPORT: All prefixes received from a BGP peer are imported into inet.0 or inet6.0.

EXPORT: All prefixes in the routing table that have been learned by BGP are exported to the router's peer, that is, so long as advertising the prefix doesn't break iBGP's rule of not advertising iBGP-learned prefixes to other iBGP peers.

Need an example? Refer to Figure 11.2.

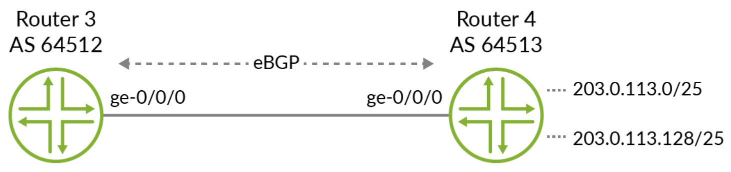


Figure 11.2 Prefixes Exported to the Router's Peer

Once again there are two routers shown in Figure 11.2 and Router 4 is advertising two prefixes to Router 3. Notice in the output below that Router 4 manipulated 203.0.113.128/25 before exporting it to Router 3, to prepend its AS-PATH three times:

```
root@Router3> show route 203.0.113.0/24

inet.0: 16 destinations, 17 routes (16 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

203.0.113.0/25     *[BGP/170] 00:00:01, localpref 100
                      AS path: 64513 I
                    > to 10.10.36.6 via ge-0/0/3.0
203.0.113.128/25   *[BGP/170] 00:00:01, localpref 100
                      AS path: 64513 64513 64513 64513 I
                    > to 10.10.36.6 via ge-0/0/3.0
```

Let's put a policy on Router 3 rejecting any prefixes that our neighbor has AS-PATH prepended. Take a look at the next configuration, but pay attention to something crucial: notice that this policy contains only one term, rejecting prepended prefixes. There isn't a second `then accept` –style statement at the end to accept everything else. Literally the only term is the one rejecting prepended prefixes:

```
set policy-options policy-statement REJECT_PREPEND from as-path REJECT_64513_PREPEND
set policy-options policy-statement REJECT_PREPEND then reject

set policy-options as-path REJECT_64513_PREPEND "64513 64513 .*"
set protocols bgp group TO_AS64513 import REJECT_PREPEND
```

And yet, when you look in the routing table now, you'll see that we're rejecting the prepended prefix and still accepting the other route:

```
root@Router3> show route 203.0.113.0/24

inet.0: 16 destinations, 17 routes (15 active, 0 holddown, 1 hidden)
+ = Active Route, - = Last Active, * = Both

203.0.113.0/25     *[BGP/170] 01:34:39, localpref 100
                      AS path: 64513 I
                    > to 10.10.36.6 via ge-0/0/3.0
```

So, there you have it: BGP has a default import action of *accept*.

OSPF and IS-IS also have a default import action of *accept*. In other words, any prefixes learned by OSPF/IS-IS will be imported into the routing table.

The default export policy often confuses people, though, because for OSPF and IS-IS the default action is to *reject*. This might seem a little counterintuitive. Surely, if a router learns a prefix by OSPF/IS-IS, it should then advertise that to its other OSPF/IS-IS neighbors?

As it happens, the router does indeed advertise these prefixes on to its neighbors, but – and here's the twist – our beautiful router doesn't do it by taking the prefixes out of the routing table and then advertising them on. Instead, it takes the LSAs (OSPF) and LSPs (IS-IS) learned from a neighbor and passes those same LSAs/LSPs on to its own neighbor.

To be clear, the prefixes are still fully being passed on, they're just not being taken from inet.0 or inet6.0. In other words, if a route is learned by OSPF/IS-IS and it's imported into the routing table, then it isn't being exported back out of the routing table again like it is with BGP. Instead the LSAs/LSPs take care of the job.

This leads to an interesting situation: if Router A advertises a prefix via OSPF or IS-IS to Router B, it isn't possible for Router B to filter that prefix out of its advertisements to Router C. Even if you configure Router B with a policy to filter a prefix out, the LSAs/LSPs will still be advertised, because the entire network has to have exactly the same link-state database. That's just the law! You can still manipulate the forwarding table on an individual router, but the advertisements themselves must be consistent across the entire domain.

TIP Let's talk about RIP in exactly as much detail as it deserves. Here we go. *Stop using RIP*. Why are you still using RIP? It's 2019. Stop it.

Discussion

LDP, RSVP, PIM - there's plenty more protocols, and each one has its own defaults. This handy link provides all the defaults for all the protocols you can put policies on: https://www.juniper.net/documentation/en_US/junos/topics/concept/policy-routing-policies-actions-defaults.html.

My grandmother always used to say "honesty is the best policy." Well, if she'd used Junos, she'd instead say that "simplicity is the best policy."

A policy with too many terms, and too many chains has more chance of going wrong, and it can be confusing to read. Always remember, complexity is the enemy of understanding. Nevertheless, the ability to make many granular decisions all in one policy makes Junos routing policy stand out from pretty much every other vendor. Use it well, and you'll be an Internet superhero.

Recipe 12: ZTP with SLAX on EX Series Devices

By Christian Scholz

- Junos Version Used: 11.x, 12.x,13.x,14.x,15.x,16.x,17.x,18.x
- Juniper Platforms General Applicability: EX Series

This recipe sets up Zero Touch Provisioning (ZTP) with the help of a SLAX script with just one single DHCP option inside your Windows server.

NOTE The script was tested on the EX3300, 3400, 2200, 2300, and 4200. It *should* run on any EX / QFX Series device, but has not yet been tested on every single model.

Problem

You want to try ZTP capabilities inside your out-of-band network, but you only have a corporate Windows DHCP Server and your trusty FTP server where you save all your configs. You don't want your server colleagues inserting different MAC-IP-S/N bindings into your DHCP server to give your new devices the needed configs. So, what now?

Solution

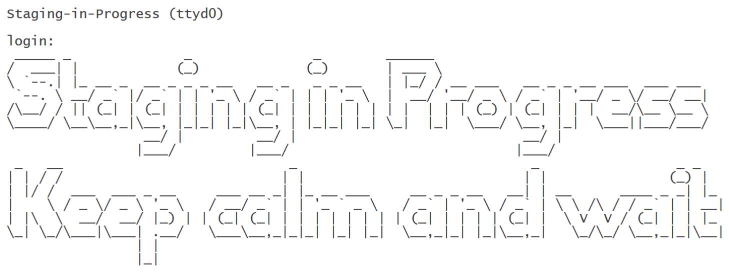
Thankfully, the Junos OS is able to run SLAX scripts. With the help of this SLAX script you will be able to stage your device (including a software update) based on its serial number (S/N). Since the S/N is transferred to you for every device you buy, you can proactively create the devices configuration and put it on your FTP server / SCP server for your switch to fetch once it reaches your department.

A 2017 Juniper script was used as a basis here, but we also added intelligence and merged another ZTP script found on the Internet to create a massive hybrid, with special thanks to Damien and Jeremy for the base scripts.

A lot of the error messages were also changed to make this script easier to follow and some new parts were added that weren't there before. You can add more and more capabilities to the script as you like.

Let's go through the script and check the steps that it uses:

```
system {
    host-name staging-in-progress;
        root-authentication {
        encrypted-password "YOUR PASSWORD"; ## SECRET-DATA
        }
             login {
        message "
```



```
Staging-in-Progress (ttyd0)
login:
```

```
                ";
          }
```

This gives the device a nice little ASCII banner "Staging in Progress" so other admins know that it is currently in staging mode:

```
    services {
        ssh;
    }
           syslog {
        user * {
            any emergency;
        }
        file messages {
            any notice;
            authorization info;
        }
    }

}
interfaces {
    me0 {
        unit 0 {
            family inet {
                dhcp;
            }
        }
    }
```

```
                vme {
        unit 0 {
            family inet {
                dhcp;
            }
        }
    }
}
```

Me0 and vme0 still do DHCP. This way they still get the “staging configuration”:

```
#-----------------------------------------
# ZTP routine - runs every 10min - The EX2200 may take its time so 10min is fine for it to "calm down"
#-----------------------------------------

system { delete: autoinstallation; }
event-options {
  generate-event { staging time-interval 600; }
  policy staging  {
    events staging ;
    then {
      execute-commands {
        commands {
          "op url ftp://ztpuser:ztppassword@10.10.10.10/staging.slax server 10.10.10.10 ex2200 
14.1X53-D40.8 ex2300 15.1X53-D55.5 ex3300 14.1X53-D40.8 ex4300 14.1X53-D40.8";
```

This Op script is executed every 10 mins, telling the device to run the staging.slax script from your FTP server, and it knows the desired version based on your script options. You can include more options in the script – and you need to do this for every new device you add in the future (more about that later):

```
          }
        }
      }
    }
}
```

The staging configuration is really easy and straightforward. Now, all you need to do is to configure your Windows server to use Option 42 to serve this staging.conf to your device. Since all the “intelligence” is inside your staging.slax script that will be loaded later, the server guys never need to touch the DHCP options again – which gives you the flexibility to change settings for your switches without involving other departments.

Let’s continue with staging.slax – our “intelligence” script:

```
/*
 *
 *
 *
 */

version 1.0;

/* Junos standard namespaces */
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

/* EXSLT extensions and app specific namespaces */
ns exsl extension = "http://exslt.org/common";
ns func extension = "http://exslt.org/functions";
ns ztp = "http://xml.juniper.net/junos/ztp";

import "/var/run/scripts/import/junos.xsl";
```

These are just some of the general definitions for SLAX to work. Best practice advises you to set up a demo area to try out new scripts or use a slax-demo.slax script (change your staging.conf to use the dev instead of the real staging.slax) so you can safely try out more variants. In general, it’s a good idea not to change anything before knowing what you need to change. The definitions, for example, need to be the same every time – do not change them – ever:

```
var $arguments = {
          <argument> {
            <name> "ex2200";
            <description> "Target release for EX2200, example: 12.3R12-S12";
          }
          <argument> {
            <name> "ex2300";
            <description> "Target release for EX2300, example: 15.1X53-D55.5";
          }
          <argument> {
            <name> "ex3300";
            <description> "Target release for EX3300, example: 12.3R12-S12 ";
          }
          <argument> {
            <name> "ex4200";
            <description> "Target release for EX4200, example: 15.1X53-D55.5";
          }
          <argument> {
            <name> "ex4300";
            <description> "Target release for ex4300, example: 12.3R12-S12";
          }
          <argument> {
            <name> "server";
            <description> "Server address";
          }
          <argument> {
            <name> "reset";
            <description> "Set to yes to reinitialize tracker(default : no)";
          }
}

param $ex2200;
param $ex2300;
param $ex3300;
param $ex4200;
param $ex4300;
param $server;
param $reset="no";
```

Feel free to add more devices (in case you use them) to the script. Remember, there are more definitions below that you also need to add, when adding a new device series:

```
var $CONFIG-PREFIX = "JUNOS-";
var $REGISTER-PREFIX = "REGISTER-";

var $VARTMP-DIR = "/var/tmp/";
var $VARLOGMSGS-FILE = "/var/log/messages";
var $TMP-DIR = "/tmp/";
var $CONFIG-DIR = "ftp://ztpuser:ztppassword@" _ $server _ "/configs/";
var $UPLOAD-DIR = "ftp://ztpuser:ztppassword@" _ $server _ "/uploads/";
var $JUNOS-DIR = "ftp://ztpuser:ztppassword@" _ $server _ "/junos/";

var $EVENT-POLICY = "staging";             /* as defined in staging.conf file */
var $SYSLOG = "external.notice";
```

Define a configuration and upload directories for your FTP server. This script uses *ztpuser* as the username and *ztppassword* as the password. The FTP server IP will be 10.10.10.10:

```
/* !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
 * !!!!!                      MAIN ENTRY                        !!!!!
 * !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! */

match / {

/* Open a connection to the device */
   var $jnx = jcs:open();


/* If RESET parameter is not equal to NO, we stop the script and reinitialize the tracker */
   if($reset != "no"){
        expr ztp:end-script( $jnx );
   }


/*Get serial number of current device */
   var $serial = ztp:serial-number( $jnx );
   var $SYSLOG_TAG = " " _ $serial _ " - ";
```

```
/* Check if script is already running,
       if it is, we stop the script
       If not, we set the tracker within the Utility mib
   */
   if( ztp:is-already-running( $jnx ) )
   {
       expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "Script already running - stopping the script" );
       <xsl:message terminate="yes">;
   }
   else{
       expr ztp:set-tracker( $jnx );
   }


/* Get device model */
   var $modele = ztp:hardware-type( $jnx );


/* Get Junos current version  */
   var $current = ztp:junos-version( $jnx );


/* Check if device type correspond to something expected  */
   if( not ($modele == "EX2200" || $modele == "EX2300" || $modele == "EX3300" || $modele == "EX4200" ||
$modele == "EX4300" )) {
```

Remember to add new strings here, when adding or modifying device types:

```
       expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "Impossible to identify current device model model or
model not implemented: ", $modele, " - aborting script" );
       expr ztp:end-script( $jnx );
   }


/* based on model type we get target version number and target version name */
   var $target-version-number = ztp:get-target-version-number( $modele, $ex2200, $ex2300, $ex3300,
$ex4200, $ex4300 );
   var $target-version-name = ztp:get-target-version-name( $modele, $target-version-number );

   expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "Device is ", $modele, " currently running on Junos ",
$current );


/*** Software version Test        ***/
/*** check if upgrade is necessary ***/
/* Compare target release to current release  */
   if( $target-version-number == "" ){
       expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "No target version defined - Aborting Script." );
       expr ztp:end-script( $jnx );
   }
   else if( $current != $target-version-number ) {
       expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "Device will be upgraded to Junos ", $target-version-
number, " ... " );
       expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "Download and upgrade in progress... (this may take some
time so grab a cup of coffee)" );
       expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "Downloading Software from ", $JUNOS-DIR, $target-
version-name  );
```

```
            var $cmd-ver = <request-package-add> { <package-name> $JUNOS-DIR _ $target-version-name;
};

            expr ztp:end-script-if-error( $jnx, $SYSLOG_TAG, jcs:execute( $jnx, $cmd-ver ) , "An
error occurred during software upgrade - Aborting Script."  );

      expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "Device upgraded successfully" );
      expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "Device will reboot now" );


      var $cmd-reb = <request-reboot>;
      expr ztp:end-script-if-error( $jnx, $SYSLOG_TAG, jcs:execute( $jnx, $cmd-reb ) , "An error
occurred during software reboot - Aborting Script."  );

      <xsl:message terminate="yes">;
  }
  else if( $current == $target-version-number ) { expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "Device is
already running the target software release - no need to upgrade" ); }
```

The device will look on your FTP server for the new software, download it, and install it. It reboots the device after the software has been added. In case something goes wrong, it throws an error message:

```
/*** Check alternate partition***/
/*** if both partition are not sync, sync them ***/





  /*** EX2300 Snapshot START ***/

/* Look for backup partition */

          if( $modele == "EX2300" ){


      expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "Starting Snapshot on EX2300" );
  var $cmd_switchsnapshot = <command> "request system snapshot" ;
  var $cmd_switchsnapshot_results = jcs:invoke( $cmd_switchsnapshot );

  if( $cmd_switchsnapshot_results//self::xnm:error ) {
    expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "Not successful" );
    expr jcs:syslog( $SYSLOG, $_SYSLOG_TAG_, "Error: ", $cmd_oldfiledel_results//self::xnm:error/
message );
    expr ztp:end-script( $jnx );
  }
  else {
  expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "Snapshotting success" );
  }

}
          else{

/* Get partition list*/
  expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "Get partition list" );
  var $cmd-snap = <get-snapshot-information> { <media>"internal"; };
  var $snapshot = jcs:execute( $jnx, $cmd-snap );
  expr ztp:end-script-if-error( $jnx, $SYSLOG_TAG, $snapshot , "An error occurred during partition
list retrieval" );
```

Taking a snapshot differs from non-ELS to ELS devices. This is reflected here. If future devices will handle this differently, you can add it here.

MORE? In general, there are so-called "non-ELS" devices (legacy devices like the EX2200) and so-called "ELS" devices (newer devices like the EX2300). If you want to know about the differences between ELS and non-ELS devices, you can find more information at: https://www.juniper.net/documentation/en_US/junos/topics/task/configuration/getting-started-els.html#id-understanding-els-configuration-statement-and-command-changes.

```
/* Look for backup partition */

   for-each($snapshot//snapshot-medium[contains(., "backup")] ) {
      var $item = .;
            var $backuppartversion = "ex-" _ $target-version-number ;
      expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "Backup partition has been found with version: ",
$item/../software-version/package[2]/package-version );
            expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "Backup partition needs to be running on version:
",  $backuppartversion );

      if( $item/../software-version/package[2]/package-version != $backuppartversion ) {

            expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "Backup partition is not up to date" );
            expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "Starting partition synchronization... (this may
take some time - why not grab another coffee?)" );

            var $cmd-req-snap = <request-snapshot> { <slice>"alternate"; };
            expr ztp:end-script-if-error( $jnx,  $SYSLOG_TAG, jcs:execute( $jnx, $cmd-req-snap ) , "An
error occurred during backup partition upgrade - Aborting Script."  );

            expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "Backup partition has been upgraded to version ",
$target-version-number );
      }
      else {
           expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "Backup partition is already running target version"
);
      }

  }
  }
```

Of course you need a clean snapshot for your backup partition. This script will take care of it:

```
/*** Save Rescue config ***/
           var $cmd-req-resc-con = <request-save-rescue-configuration>;
           expr ztp:end-script-if-error( $jnx,  $SYSLOG_TAG, jcs:execute( $jnx, $cmd-req-resc-con ) ,
"An error occurred during the writing of the rescue-config - Aborting Script."  );
```

It's always a good idea to have a *rescue configuration* to get rid of the amber error LED on your device. After the final configuation has been applied, this step is executed again so that the final configuation and rescue configuation are the same:

```
/*** Specific action for each platform ***/

          if( $modele == "EX3300" ){

/* Uplink reconfiguration  */
      var $cmd-int0 = <request-virtual-chassis-vc-port-delete-pic-slot> { <pic-slot> "1"; <port>"0";
};
      var $res-int0 = jcs:execute( $jnx, $cmd-int0 );
      expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "Interface xe-0/1/0 has been converted to a standard port
- VC disabled" );

      var $cmd-int1 = <request-virtual-chassis-vc-port-delete-pic-slot> { <pic-slot> "1"; <port>"1";
};
      var $res-int1 = jcs:execute( $jnx, $cmd-int1 );
      expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "Interface xe-0/1/1 has been converted to a standard port
- VC disabled" );

      var $cmd-int2 = <request-virtual-chassis-vc-port-delete-pic-slot> { <pic-slot> "1"; <port>"2";
};
      var $res-int2 = jcs:execute( $jnx, $cmd-int2 );
      expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "Interface xe-0/1/2 has been converted to a standard port
- VC disabled" );

      var $cmd-int3 = <request-virtual-chassis-vc-port-delete-pic-slot> { <pic-slot> "1"; <port>"3";
};
      var $res-int3 = jcs:execute( $jnx, $cmd-int3 );
      expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "Interface xe-0/1/3 has been converted to a standard port
- VC disabled" );
          }
          else{
      expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "This Switchtype does not get any device-specific config
- skipping this step" );
   }
```

Sometimes you want to perform more actions, depending on your device series type. In this case, the EX3300 VC ports should be disabled:

```
/*** START Copying Config on Switch ***/

  var $config-file = $CONFIG-DIR _ $serial _ "/" _ $serial _ ".conf";
  var $dst-file = "/var/tmp/";
  var $config-file-local = $dst-file _ $serial _ ".conf";


/*** START Deleting old local config files ***/
```

```
   expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "Deleting old configs from /var/tmp/" );
   var $cmd_oldfiledel = <command> "file delete " _ $config-file-local ;
   var $cmd_oldfiledel_results = jcs:invoke( $cmd_oldfiledel );

   if( $cmd_oldfiledel_results//self::xnm:error ) {
      expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "Unable to delete old config from /var/tmp/" );
      expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "Error: ", $cmd_oldfiledel_results//self::xnm:error/
message );
      expr ztp:end-script( $jnx );
   }

   else {
   expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "Old config deletes successfully from /var/tmp/" );
   }


/*** END Deleting old local config files ***/

   expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "Now copying the final config from the Server to the Switch"
);
   expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "Copying ", $config-file, " to ", $dst-file );
   var $cpy-finalconf = <file-copy> { <source> $config-file; <destination> $dst-file; }
   var $cpy-finalconfigexec = jcs:execute( $jnx, $cpy-finalconf );



   if( $cpy-finalconfigexec//self::xnm:error ) {
            expr ztp:end-script-if-error( $jnx,  $SYSLOG_TAG, jcs:execute( $jnx, $cpy-finalconfigexec
) , "ERROR: Could not fetch my final config. See error below"  );
      call syslog-messages( $header, $messages = $cpy-finalconfigexec//self::xnm:error/message );
   }
   else {

            expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "Fetched final config and saved it to ", $config-
file-local );

   }

/*** END START Copying Config on Switch ***/


  <op-script-results> {
/* Lock the configuration */
             expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "Locking the config for commit" );
        var $lock-results = jcs:execute( $jnx, "lock-configuration" );
        if( jcs:empty( $lock-results/..//xnm:error ) ) {

            var $load-rpc = <load-configuration url=$config-file-local action="override" format="text">;
                 expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "Committing final-config - please wait..." );
            var $load-results = jcs:execute( $jnx, $load-rpc );

            if( jcs:empty( $load-results/..//self::xnm:error ) ) {

                var $commit-results = jcs:execute( $jnx, "commit-configuration" );
                copy-of $commit-results;
                var $unlock-results = jcs:execute( $jnx, "unlock-configuration" );
                     expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "Committed - now unlocking config" );
                copy-of $unlock-results;
```

```
                  expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "Unlocking of config successful!" );
            }
            else {
               copy-of $load-results;

/* Don't leave it locked because the load failed */
               var $unlock-results = jcs:execute( $jnx, "unlock-configuration" );
               copy-of $unlock-results;
            }
         }
         else {
            copy-of $lock-results;
         }
      }
   expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "New Device config is now active" );
```

You now have your config applied to the device. At this time you could power off your device manually and everything would be fine – but we want to finish it the automated way, right?

```
/* Write file with S/N to Upload-DIR */

   expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "Uploading /var/log/messages to Server" );
   var $cpy-upload = <file-copy> { <source> $VARLOGMSGS-FILE ; <destination> $UPLOAD-DIR _ $serial _
".log" ; }
   var $cpy-upload-configdone = jcs:execute( $jnx, $cpy-upload );


   if( $cpy-upload-configdone//self::xnm:error ) {
      expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "Unable to Upload file!" );
      expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "Error: ", $cpy-upload-configdone//self::xnm:error/
message );
      expr ztp:end-script( $jnx );
   }

   else {
   expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "Upload successfully done!" );
   }


   expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "Staging finished - initiating Shutdown of Switch..." );


  var $cmd_switchshut = <command> "request system power-off in 0";
  var $cmd_switchshut_results = jcs:invoke( $cmd_switchshut );


   expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "Shutting down. Thank you for staging with us - have a nice
day." );


   expr ztp:end-script( $jnx );
```

The switch will now upload the log files from staging (written to the messages log) to your FTP server, and after that, it will shut down. Once this happens you know that your device has been successfully staged. And since your final configuration does not include the Op script, a *scripting loop* is prevented.

What follows is the *under the hood* section, where you can tweak your script and add new device types:

```
}

/* !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
 * !!!!!          Under the Hood - please do not change until you know exactly what you are doing and 
have at least mastered SLAX             !!!!!
 * !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! */




/* !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
 * !!!!!          "Helper" Functions/Templates             !!!!!
 * !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! */

<func:function name="ztp:ping">
{
  param $jnx;
  param $_host_;
  param $_source_;

  var $cmd = <ping> { <host> $_host_;
    <no-resolve>;
    <source> $_source_;
    <count> 1;
  }

  var $rsp = jcs:execute( $jnx, $cmd );

  <func:result select="exsl:node-set($rsp)">;
}

/* ----------------------------------------------------------------------- */
/* this function is used to return the right target junos version number */
/* ----------------------------------------------------------------------- */
<func:function name="ztp:get-target-version-number">
{
  param $_modele_;
  param $_ex2200_;
  param $_ex2300_;
  param $_ex3300_;
  param $_ex4200_;
  param $_ex4300_;

  if( $_modele_ == "EX2200" ) { <func:result select="string($_ex2200_)">; }
  else if( $_modele_ == "EX2300" ) { <func:result select="string($_ex2300_)">; }
  else if( $_modele_ == "EX3300" ) { <func:result select="string($_ex3300_)">; }
  else if( $_modele_ == "EX4200" ) { <func:result select="string($_ex4200_)">; }
  else if( $_modele_ == "EX4300" ) { <func:result select="string($_ex4300_)">; }

}

/* ------------------------------------------------------------ */
/* this function is used to return the right target junos name */
/* ------------------------------------------------------------ */
```

```
<func:function name="ztp:get-target-version-name">
{
   param $_modele_;
   param $_target-version_;

   var $JUNOS-EX2200 = "jinstall-ex-2200-" _ $_target-version_ _ "-domestic-signed.tgz";
   var $JUNOS-EX2300 = "junos-arm-32-" _ $_target-version_ _ ".tgz";
   var $JUNOS-EX3300 = "jinstall-ex-3300-" _ $_target-version_ _ "-domestic-signed.tgz";
   var $JUNOS-EX4200 = "jinstall-ex-4200-" _ $_target-version_ _ "-domestic-signed.tgz";
   var $JUNOS-EX4300 = "jinstall-ex-4300-" _ $_target-version_ _ "-domestic-signed.tgz";

   if( $_modele_ == "EX2200" ) { <func:result select="string($JUNOS-EX2200)">; }
   else if( $_modele_ == "EX2300" ) { <func:result select="string($JUNOS-EX2300)">; }
   else if( $_modele_ == "EX3300" ) { <func:result select="string($JUNOS-EX3300)">; }
   else if( $_modele_ == "EX4200" ) { <func:result select="string($JUNOS-EX4200)">; }
   else if( $_modele_ == "EX4300" ) { <func:result select="string($JUNOS-EX4300)">; }

}

/* -------------------------------------------------- */
/*   this function is used get the software version  */
/* -------------------------------------------------- */
<func:function name="ztp:junos-version">
{
   param $jnx;

   var $cmd = <get-configuration> { <configuration> { <version>; }}
   var $ver = jcs:execute( $jnx, $cmd )//version;

   <func:result select="$ver">;
}
/* -------------------------------------------------- */
/*     this function is used get hardware type     */
/* -------------------------------------------------- */
<func:function name="ztp:hardware-type">
{
   param $jnx;
   var $cmd = <get-chassis-inventory>;
   var $res = jcs:execute( $jnx, $cmd );
   var $model-full = $res/chassis/description;
   var $model = jcs:split("-", $model-full);
   <func:result select="$model[1]">;
}

/* -------------------------------------------------- */
/*     this function is used get serial number     */
/* -------------------------------------------------- */
<func:function name="ztp:serial-number">
{
   param $jnx;
   var $cmd = <get-chassis-inventory>;
   var $res = jcs:execute( $jnx, $cmd );
   var $model-full = $res/chassis/serial-number;
   <func:result select="$model-full[1]">;
}

/* --------------------------------------------------- */
/* this function is used to set activity tracker to 1 */
/* --------------------------------------------------- */
```

```
<func:function name="ztp:set-tracker">
{
   param $jnx;

   var $cmd = <request-snmp-utility-mib-set> {
                  <object-type> "integer";
                  <instance> "staging";
                  <object-value> "1";
              }

   var $res = jcs:execute( $jnx, $cmd );

   <func:result select="true()">;
}

/* ---------------------------------------------------- */
/* this function is used to set activity tracker to 0 */
/* ---------------------------------------------------- */
<func:function name="ztp:remove-tracker">
{
   param $jnx;

   var $cmd = <request-snmp-utility-mib-set> {
                  <object-type> "integer";
                  <instance> "staging";
                  <object-value> "0";
              }

   var $res = jcs:execute( $jnx, $cmd );

   <func:result select="true()">;
}

/* --------------------------------------------------- */
/*   this function is used to get activity tracker   */
/* --------------------------------------------------- */
<func:function name="ztp:is-already-running">
{
   param $jnx;

   var $cmd = <get-snmp-object> {
                        <snmp-object-
name>"1.3.6.1.4.1.2636.3.47.1.1.3.1.2.106.101.97.112.45.100.101.105";
                }

   var $res = jcs:execute( $jnx, $cmd )//snmp-object/object-value;

   if( $res == "1" ){ <func:result select="true()">; }
   else{ <func:result select="false()">; }
}

/* ----------------------------------------------------- */
/* this function is used to terminate script properly  */
/* ----------------------------------------------------- */
```

```
<func:function name="ztp:end-script">
{
   param $jnx;

   var $res = ztp:remove-tracker($jnx);
   expr jcs:close( $jnx );
   <xsl:message terminate="yes">;

   <func:result select="true()">;
}

/* ------------------------------------------------- */
/* this function is used to test error              */
/* and terminate script properly if an error happens */
/* ------------------------------------------------- */
<func:function name="ztp:end-script-if-error">
{
   param $jnx;
   param $_SYSLOG_TAG_;
   param $_result_;
   param $_log_;

   if( $_result_//self::xnm:error ) {
      expr jcs:syslog( $SYSLOG, $_SYSLOG_TAG_, $_log_, " - STOP" );
      expr jcs:syslog( $SYSLOG, $_SYSLOG_TAG_, "ERROR : ", $_result_//self::xnm:error/message );
      expr ztp:end-script( $jnx );
   }

   <func:result select="true()">;
}

/* -------------------------------------------------------------------------------- */
/* this function is used to load/commit changes to the configuration; no locking */
/* -------------------------------------------------------------------------------- */
<func:function name="ztp:load-config">
{
   param $jnx;
   param $load-cmd;

   var $load-rsp = jcs:execute( $jnx, $load-cmd );
   if( $load-rsp//self::xnm:error ) {
      call syslog-messages( $header = "Unable to load configuration", $messages = $load-rsp//
self::xnm:error/message );
      <func:result select="$load-rsp//self::xnm:error">;
   }
   else {
      var $commit-rsp = jcs:execute( $jnx, "commit-configuration" );
      if( $commit-rsp//self::xnm:error ) {
         call syslog-messages( $header = "unable to commit configuration", $messages = $commit-rsp//
self::xnm:error/message );
         <func:result select="exsl:node-set($commit-rsp//self::xnm:error)">;
      }
      else {
         <func:result select="/null">;
      }
   }
}
```

```
/* ------------------------------------------------------------ */
/* this template is used to dump warning/error messages to syslog */
/* ------------------------------------------------------------ */
template syslog-messages( $header, $messages )
{
  expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, $header );
  for-each( $messages ) {
    expr jcs:syslog( $SYSLOG, $SYSLOG_TAG, "message[ ", ., "]");
  }
}
```

Discussion

Hopefully this script can help you to get started with SLAX on Junos and help you to build your awesome automation for staging. As you can see, you can massively enhance and manipulate this script the way you need to (for example create VCs with it, or even complete VCFs with predefined VC ports).

A huge shout-out to my customers (especially Lukas) for letting me code this and build another awesome Junos automation script. SLAX might look scary at first – but the more you work with it, the cooler it gets.

Recipe 13: Configuring NAT on SRX Platforms Using Proxy ARP/ND

By Yasmin Lara

Junos OS Used: 17.1R2

Juniper Platforms General Applicability: vSRX, SRX

On the SRX series, you can configure source NAT, destination NAT, and static NAT in many different ways, as shown in Figure 13.1.

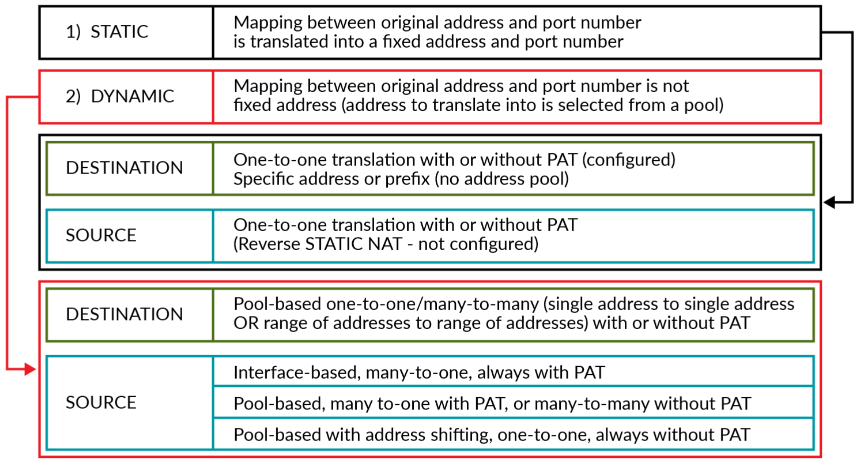

1) STATIC
Mapping between original address and port number is translated into a fixed address and port number
2) DYNAMIC
Mapping between original address and port number is not fixed address (address to translate into is selected from a pool)
DESTINATION
One-to-one translation with or without PAT (configured) Specific address or prefix (no address pool)
SOURCE
One-to-one translation with or without PAT (Reverse STATIC NAT - not configured)
DESTINATION
Pool-based one-to-one/many-to-many (single address to single address OR range of addresses to range of addresses) with or without PAT
SOURCE
Interface-based, many-to-one, always with PAT
Pool-based, many to-one with PAT, or many-to-many without PAT
Pool-based with address shifting, one-to-one, always without PAT

Figure 13.1 Options for Configuring NAT in Juniper SRX Devices

When configuring source or destination NAT, using an address pool, or a specific address (not interface-based), you can configure the SRX to translate into an address that belongs to the same address range (network) that the address of the SRX's own interface belongs to. In order for this to work, you need to configure proxy ARP or proxy NDP (for IPv4 and IPv6).

This recipe focuses on how to configure the cases that requires proxy ARP/ND.

Problem

When you configure NAT to translate either the source or the destination address of packets into an address in the same range (network) that the inbound/outbound interface belongs to, a directly connected host in that network will assume that the destination it is trying to reach, or trying to reply to (the translated address), is also on that same network.

For example, consider the situation illustrated in Figure 13.2. Interface ge-0/0/3 on vSRX1 is configured with the address 200.1.1.254/24. HostB is connected to the same network and has the address 200.1.1.20. The vSRX1 is configured to perform destination/static NAT, and to translate the destination address 200.1.1.1 of incoming packets (from zone UNTRUST to zone TRUST), into destination address 10.1.1.1 (the real IP address of HostA). The address 200.1.1.1 represents HostA as a public host that can be reached from the outside (UNTRUST zone).

When HostB has traffic to send to HostA, it assumes that Host A is local because the address that it believes HostA has (200.1.1.1) belongs to the same network (2001.1.0/24) that its own address belongs to. Thus, HostB sends an ARP request asking 200.1.1.1 to reply with its MAC address. Because 200.1.1.1 is not really a host in the network, vSRX1 just discards the ARP request, and no reply is sent back. As a result, HostB will not be able to send any traffic to HostA.

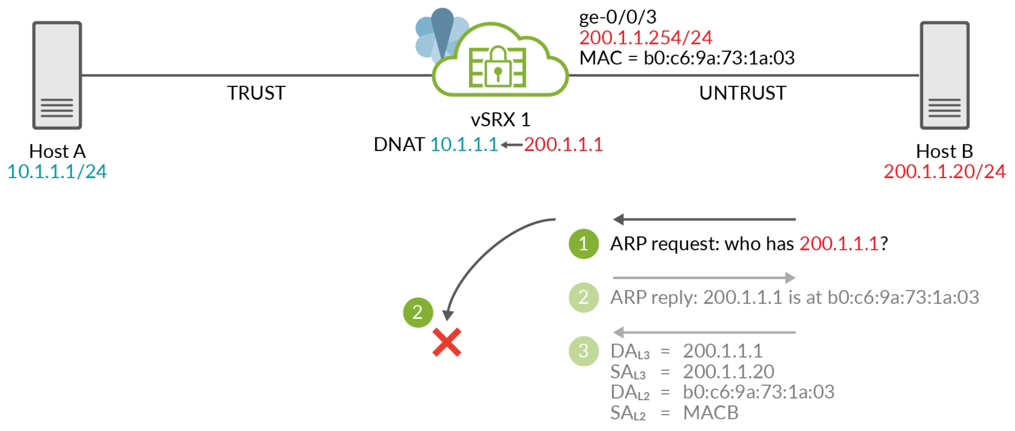


Figure 13.2 Problem with Destination/Static NAT Using an Address in the Same Range as the Inbound Interface

The same problem will be seen if vSRX1 is doing source NAT and is translating source address 10.1.1.1 into 200.1.1.1, as illustrated in Figure 13.3. In this case, the messages from HostA will reach HostB, but HostB will not be able to reply, because it will not be able to resolve 200.1.1.1 into a MAC address, as described in the previous case (destination/source NAT).

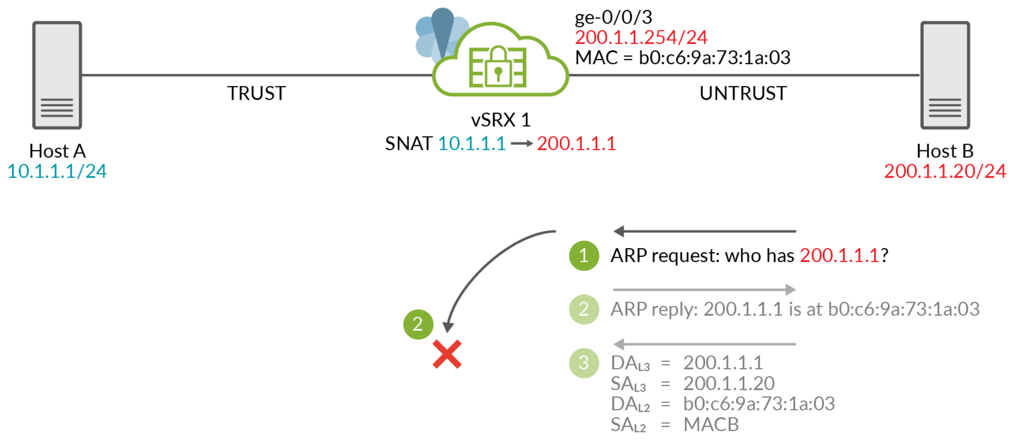


Figure 13.3 *Problem with Source NAT Using an Address in the Same Range as the Outbound Interface*

You will also see this behavior when you are doing IPv6/IPv4 translations. As an example, consider the situation illustrated in Figure 13.4. vSRX1 is doing IPv6 to IPv4 address translation, and it is configured to translate destination address 2001:db8:1:1::1 into 10.1.1.1, and source address 2001:db8:1:1::20 into 10.1.1.20.

When HostB has traffic to send to HostA, it assumes that HostA is on the same network and sends a neighbor solicitation message asking 2001:db8:1:1::1 to reply with its MAC address. However, because 2001:db8:1:1::1 is not really a host in the network, vSRX1 will discard the NS and no reply will be sent back. Again, HostB will not be able to send any traffic to HostA.

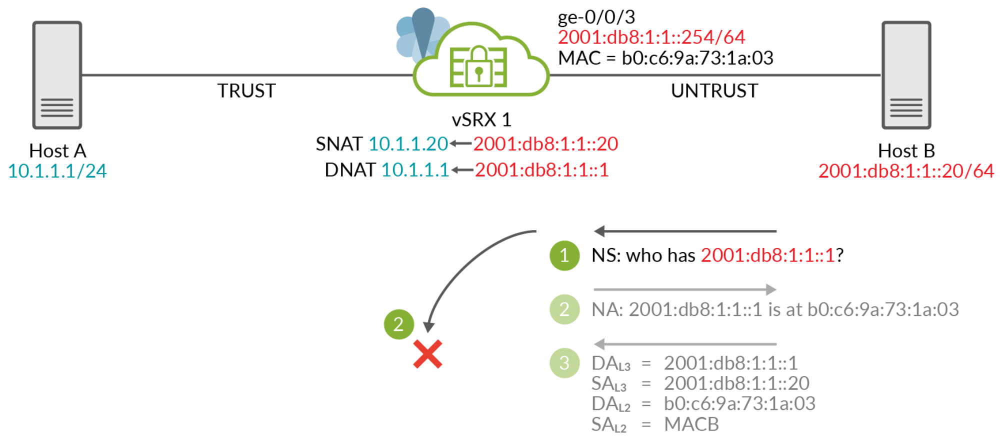


Figure 13.4 *Problem with Source and Destination IPv6 to IPv4 Translation*

Solution

The solution to the problem described is to configure proxy ARP and/or proxy NDP on the interface(s) where the ARP request(s) will be received.

When you configure proxy ARP or proxy NDP, the vSRX responds to ARP requests, using its own MAC address. You can see how this works in Figures 13.5 and 13.6.

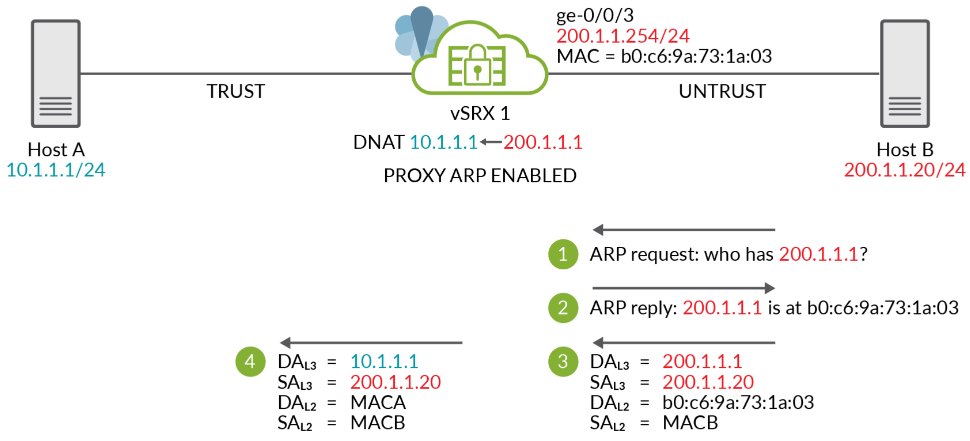


Figure 13.5 Proxy ARP Enabled

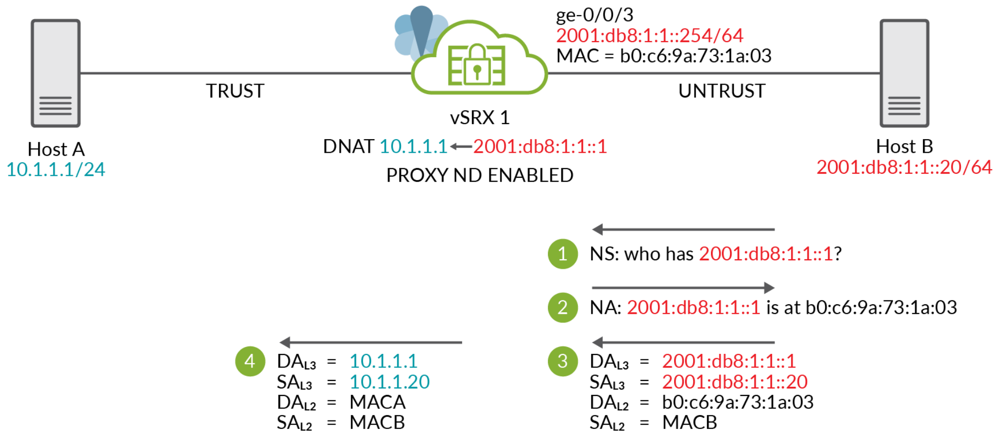


Figure 13.6 Proxy ND Enabled

To demonstrate how proxy ARP and proxy NDP work, and how to configure them, this recipe uses the topology shown in Figure 13.7.

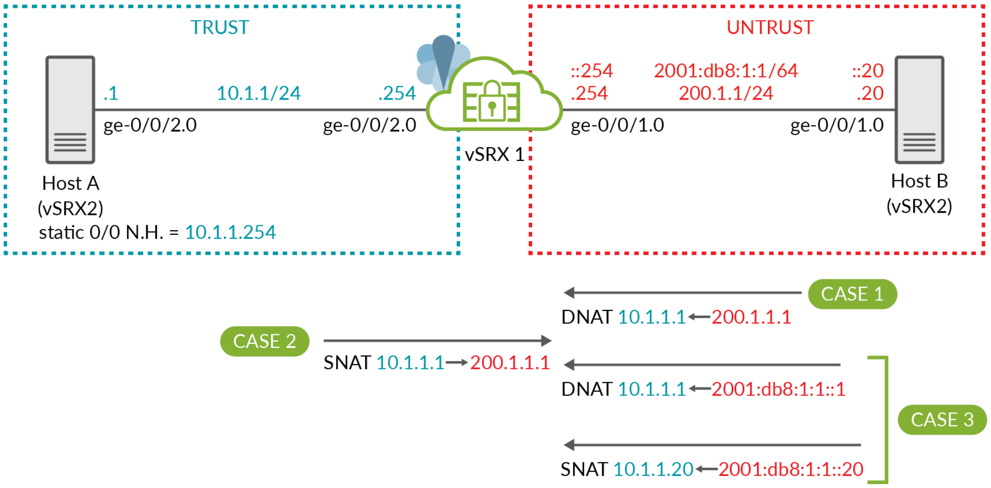


Figure 13.7 Topology for Recipe 13

The recipe looks at three cases:

- Destination NAT
- Source NAT
- IPv6 to IPv4 source and destination NAT

NOTE vSRX2 is configured to perform packet-mode processing, and it is already configured with two routing instances that simulate HostA and HostB. This configuration is not covered in this recipe.

General Configuration Steps

Configure the interfaces:

```
[edit interfaces ge-0/0/1 unit 0]
lab@vSRX-1# show
family inet {
   address 200.1.1.254/24;
}
family inet6 {
   address 2001:db8:1:1::254/64;
}

[edit interfaces ge-0/0/1 unit 0]
lab@vSRX-1# show | display set
set interfaces ge-0/0/1 unit 0 family inet address 200.1.1.254/24
set interfaces ge-0/0/1 unit 0 family inet6 address 2001:db8:1:1::254/64

[edit interfaces ge-0/0/2]
lab@vSRX-1# show
```

```
unit 0 {
   family inet {
      address 10.1.1.254/24;
   }
}

[edit interfaces ge-0/0/2]
lab@vSRX-1# show| display set
set interfaces ge-0/0/2 unit 0 family inet address 10.1.1.254/24
```

Configure the security zones:

```
[edit security zones]
lab@vSRX-1# show
security-zone TRUST {
   host-inbound-traffic {
      system-services {
         ping;
      }
   }
   interfaces {
      ge-0/0/2.0;
   }
}
security-zone UNTRUST {
   host-inbound-traffic {
      system-services {
         ping;
      }
   }
   interfaces {
      ge-0/0/1.0;
   }
}

[edit security zones]
lab@vSRX-1# show | display set
set security zones security-zone TRUST host-inbound-traffic system-services ping
set security zones security-zone TRUST interfaces ge-0/0/2.0

set security zones security-zone UNTRUST host-inbound-traffic system-services ping
set security zones security-zone UNTRUST interfaces ge-0/0/1.0
```

Verify that you can reach HostA and HostB from the vSRX:

```
[edit]
lab@vSRX-1# run ping 10.1.1.1 rapid
PING 10.1.1.1 (10.1.1.1): 56 data bytes
!!!!!
--- 10.1.1.1 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max/stddev = 3.667/4.243/4.967/0.520 ms

[edit]
lab@vSRX-1# run ping 200.1.1.20 rapid
PING 200.1.1.20 (200.1.1.20): 56 data bytes
!!!!!
--- 200.1.1.20 ping statistics ---
```

```
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max/stddev = 1.897/9.845/21.910/7.923 ms
```

Configure entries in the global address book:

```
[edit security address-book global]
lab@vSRX-1# show
address HOSTA-10.1.1.1/32 {
  description "Actual IPv4 address of HostA";
    10.1.1.1/32;
}
address HOSTA-200.1.1.1/32 {
    description "IPv4 address by which HostA is known by hosts in the UNTRUST Zone";
    200.1.1.1/32;
}
address HOSTA-IPv6 {
    description "IPv6 address by which HostA is known by hosts in the UNTRUST Zone;";
    2001:db8:1:1::1/128;
}
address HOSTB-IPv6 {
    description "Actual IPv6 address of HostB ";
    2001:db8:1:1::20/128;
}
address HOSTB-10.1.1.20/32 {
    description "IPv4 address by which HostB is known by hosts in the UNTRUST Zone";
    10.1.1.20/32;
}
address HOSTB-200.1.1.20 {
    description "Actual IPv4 address of HostB;";
    200.1.1.20/32;
}
```

NOTE Make sure that IPv6 processing is enabled before creating entries in the address book with IPv6 addresses. This might require a system reboot depending on the version of Junos running in your device.

```
[edit security address-book global]
lab@vSRX-1# show | display set relative
set address HOSTA-10.1.1.1/32 description "Actual IPv4 address of HostA"
set address HOSTA-10.1.1.1/32 10.1.1.1/32
set address HOSTA-200.1.1.1/32 description "IPv4 address by which HostA is known by hosts in the
UNTRUST Zone"
set address HOSTA-200.1.1.1/32 200.1.1.1/32
set address HOSTA-IPv6 description "IPv6 address by which HostA is known by hosts in the UNTRUST Zone;"
set address HOSTA-IPv6 2001:db8:1:1::1/128
set address HOSTB-IPv6 description "Actual IPv6 address of HostB "
set address HOSTB-IPv6 2001:db8:1:1::20/128

set address HOSTB-10.1.1.20/32 description "IPv4 address by which HostB is known by hosts in the TRUST
Zone"
set address HOSTB-10.1.1.20/32 10.1.1.20/32

set address HOSTB-200.1.1.20 description "Actual IPv4 address of HostB;"
set address HOSTB-200.1.1.20 200.1.1.20/32
```

IPv4 Destination NAT and Proxy ARP

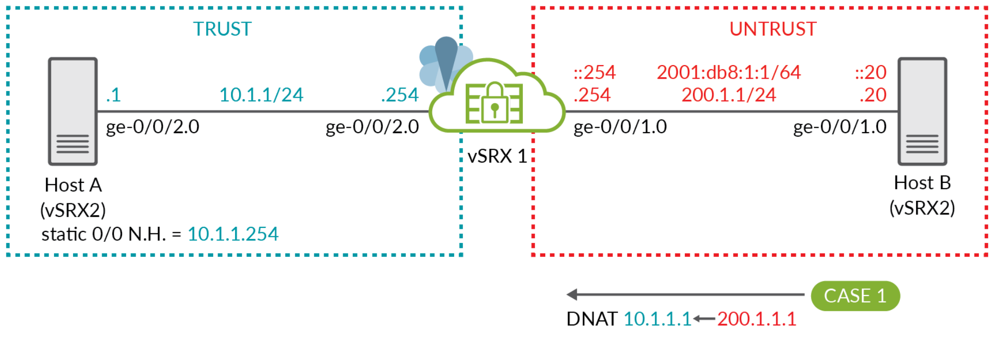


Figure 13.8 Case #1 – Destination NAT

Create a `rule-set` for destination NAT, and specify where the traffic to be translated is coming from:

```
 [edit security nat destination]
lab@vSRX-1# show
rule-set from-untrust-dest-rule-set {
    from zone UNTRUST;
}

[edit security nat destination]
lab@vSRX-1# show | display set relative
set rule-set from-untrust-dest-rule-set from zone UNTRUST
```

NOTE For destination and static NAT configuring the `from` statement and matching on `destination-address` are mandatory. The following table summarizes the requirements.

TYPE	RULE-SET / RULE REQUIRED MATCHING CRITERIA	
	DIRECTION (RULE-SET)	PACKET INFO. (RULE)
STATIC or DESTINATION NAT	`from` (zone, routing-instance, or interface)	`destination-address`
SOURCE NAT	`from` (zone, routing-instance, or interface) AND `to` (zone, routing-instance, or interface)	`source-address` OR `destination-address` (at least one)

Create a translation rule within the `rule-set`, that matches the destination address 200.1.1.1/32, and translates it into 10.1.1.1/32:

```
[edit security nat destination rule-set from-untrust-dest-rule-set ]
lab@vSRX-1# show
from zone UNTRUST;
```

```
rule DNAT-200-to-10 {
   match {
      destination-address 200.1.1.1/32;
   }
   then {
      destination-nat {
         pool {
            DNAT-POOL-10;
         }
      }
   }
}

[edit security nat destination rule-set from-untrust-dest-rule-set ]
lab@vSRX-1# show | display set relative
set from zone UNTRUST
set rule DNAT-200-to-10 match destination-address 200.1.1.1/32
set rule DNAT-200-to-10 then destination-nat pool DNAT-POOL-10

[edit security nat destination pool DNAT-POOL-10]
lab@vSRX-1# show
address 10.1.1.1/32;

[edit security nat destination pool DNAT-POOL-10]
lab@vSRX-1# show | display set
set security nat destination pool DNAT-POOL-10 address 10.1.1.1/32
```

Configure a security policy that allows HostB to reach HostA using any application:

```
[edit security policies from-zone UNTRUST to-zone TRUST]
lab@vSRX-1# show policy HostB-to-HostA
   match {
      source-address HOSTB-200.1.1.20;
      destination-address HOSTA-10.1.1.1/32;
      application any;
   }
   then {
      permit;
      log {
         session-init;
         session-close;
      }
   }

[edit security policies from-zone UNTRUST to-zone TRUST]
lab@vSRX-1# show | display set relative
set policy HostB-to-HostA match source-address HOSTB-200.1.1.20
set policy HostB-to-HostA match destination-address HOSTA-10.1.1.1/32
set policy HostB-to-HostA match application any
set policy HostB-to-HostA then permit
set policy HostB-to-HostA then log session-init
set policy HostB-to-HostA then log session-close
```

NOTE When writing the policy we used the translated destination address of HostA.

Static and destination NAT are applied before the policy is applied to the traffic, as shown in Figure 13.9. Therefore, you need to make sure to match on the translated destination address (and port number if applicable).

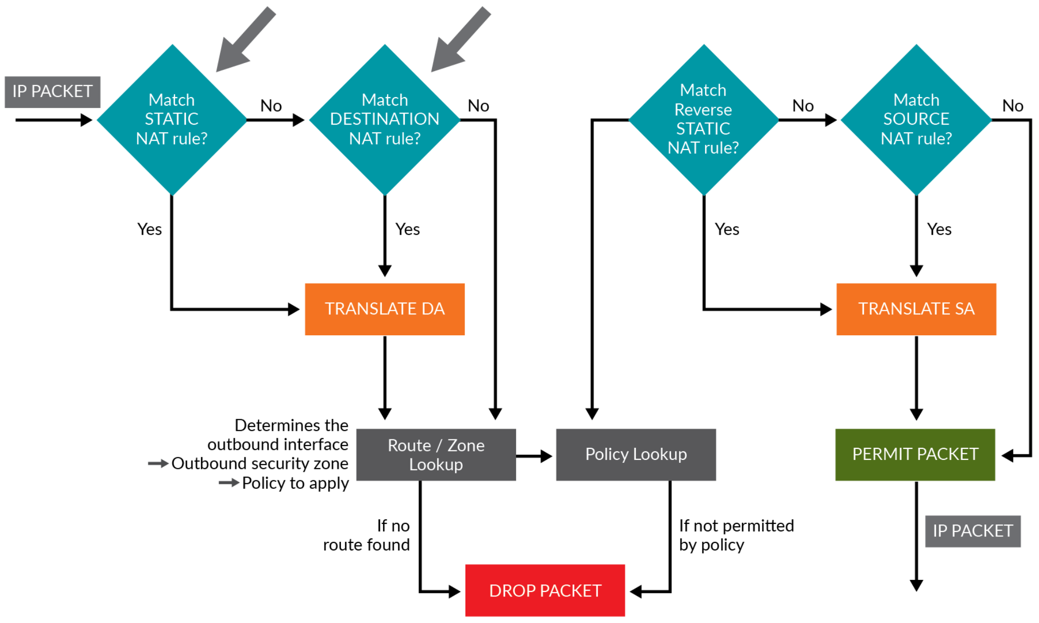


Figure 13.9 Packet Processing With NAT

For example, if you are translating:

```
DA = 208.1.1.1 & Port = 21, to
DA = 10.1.1.1 & Port = 2121 (custom-ftp)

Your policy should match:

destination address = 10.1.1.1
source address = any
application = 2121 (custom-ftp)
```

NOTE The policy from zone UNTRUST to zone TRUST configured in this step, includes the actions `log session-init` and `log session-close`, but you still need to configure logging under system syslog, with the proper facility and severity level, to capture the messages generated when the sessions are created and terminated. This is shown in the next step.

Configure logging so that you can see the session creation and termination.

One easy way of doing this is to create a file with facility and severity level `any`, and then add a filter that matches on the tag `RT_FLOW`:

```
[edit system syslog]
lab@vSRX-1# show
file SECURITY {
   any any;
   match RT_FLOW;
}

[edit system syslog]
lab@vSRX-1# show | display set relative
set file SECURITY any any
set file SECURITY match RT_FLOW
```

Verify that HostB can communicate with HostA.

After configuring the interfaces, security zones, and the policies we can try sending traffic from HostB to HostA, but we will see that it fails:

```
lab@vSRX-2> ping 200.1.1.1 routing-instance HOSTB
PING 200.1.1.1 (200.1.1.1): 56 data bytes
^C
--- 200.1.1.1 ping statistics ---
169 packets transmitted, 0 packets received, 100% packet loss
```

We can also check if any flow session was created:

```
lab@vSRX-1> show security flow session protocol icmp
Total sessions: 0
```

But we find none.

Also, if we monitor traffic on interface ge-0/0/1.0 on vSRX2, we can see that HostB is attempting to resolve the IP addresses of HostA but is not getting any responses:

```
lab@vSRX-2> monitor traffic interface ge-0/0/1.0
verbose output suppressed, use <detail> or <extensive> for full protocol decode
Address resolution is ON. Use <no-resolve> to avoid any reverse lookup delay.
Address resolution timeout is 4s.
Listening on ge-0/0/1.0, capture size 96 bytes

Reverse lookup for 200.1.1.1 failed (check DNS reachability).
Other reverse lookup failures will not be reported.
Use <no-resolve> to avoid reverse lookups on IP addresses.

16:05:39.197930 Out arp who-has 200.1.1.1 tell 200.1.1.20
16:05:40.100052 Out arp who-has 200.1.1.1 tell 200.1.1.20
16:05:41.011853 Out arp who-has 200.1.1.1 tell 200.1.1.20
^C
4 packets received by filter
0 packets dropped by kernel
```

Configure proxy ARP on interface ge-0/0/1.0 in vSRX1 to respond to ARP requests for 200.1.1.1.

Add `proxy-arp` under edit security nat and commit the configuration:

```
[edit security nat proxy-arp]
lab@vSRX-1# show
interface ge-0/0/1.0 {
   address {
      200.1.1.1/32;
   }
}

[edit security nat proxy-arp]
lab@vSRX-1# show | display set
set security nat proxy-arp interface ge-0/0/1.0 address 200.1.1.1/32

 [[edit security nat proxy-arp]
lab@vSRX-1# commit
commit complete
```

As soon as the new vSRX1 configuration is activated, we see that an ARP response is sent to HostB and traffic starts flowing from HostB to HostA:

```
lab@vSRX-2> monitor traffic interface ge-0/0/1.0
verbose output suppressed, use <detail> or <extensive> for full protocol decode
Address resolution is ON. Use <no-resolve> to avoid any reverse lookup delay.
Address resolution timeout is 4s.
Listening on ge-0/0/1.0, capture size 96 bytes

Reverse lookup for 200.1.1.1 failed (check DNS reachability).
Other reverse lookup failures will not be reported.
Use <no-resolve> to avoid reverse lookups on IP addresses.

---more---
16:09:32.120438 Out arp who-has 200.1.1.1 tell 200.1.1.20
16:09:32.721456 Out arp who-has 200.1.1.1 tell 200.1.1.20
16:09:32.725402  In arp reply 200.1.1.1 is-at 2c:c2:60:5a:09:b2
---more---
16:09:38.602751 Out IP truncated-ip - 24 bytes missing! 200.1.1.20 > 200.1.1.1: ICMP echo request, id 
34715, seq 301, length 64
16:09:38.604679  In IP 200.1.1.1 > 200.1.1.20: ICMP echo reply, id 34715, seq 301, length 64
^C
39 packets received by filter
0 packets dropped by kernel
```

vSRX1 is responding to the ARP request using its own MAC address:

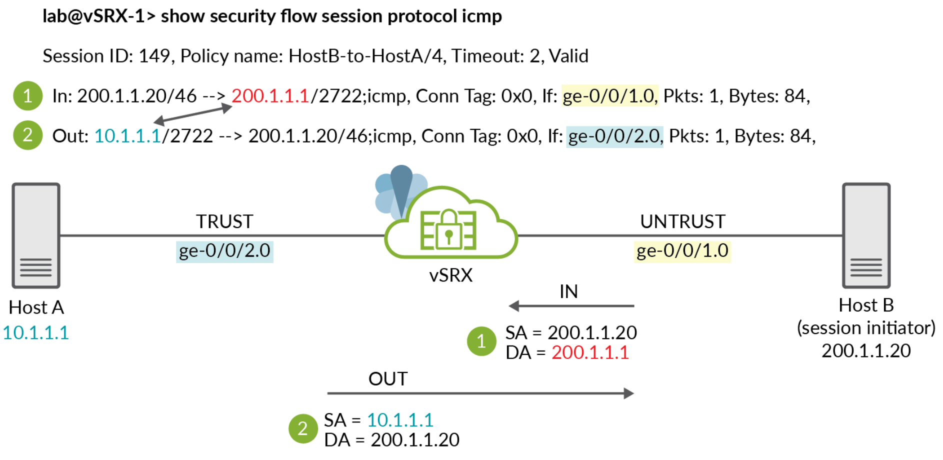
```
lab@vSRX-1> show interfaces ge-0/0/1 extensive | match hardw
  Current address: 2c:c2:60:5a:09:b2, Hardware address: 2c:c2:60:5a:09:b2

lab@vSRX-2> show arp interface ge-0/0/1.0
MAC Address       Address         Name          Interface     Flags
2c:c2:60:5a:09:b2 200.1.1.1       200.1.1.1     ge-0/0/1.0    none
2c:c2:60:5a:09:b2 200.1.1.254     200.1.1.254   ge-0/0/1.0    none
Total entries: 2

lab@vSRX-2> ping 200.1.1.1 routing-instance HOSTB
PING 200.1.1.1 (200.1.1.1): 56 data bytes
---more---
64 bytes from 200.1.1.1: icmp_seq=296 ttl=63 time=2.816 ms
64 bytes from 200.1.1.1: icmp_seq=297 ttl=63 time=8.573 ms
64 bytes from 200.1.1.1: icmp_seq=298 ttl=63 time=2.958 ms
```

```
64 bytes from 200.1.1.1: icmp_seq=299 ttl=63 time=3.066 ms
64 bytes from 200.1.1.1: icmp_seq=300 ttl=63 time=2.933 ms
64 bytes from 200.1.1.1: icmp_seq=301 ttl=63 time=4.998 ms
^C
--- 200.1.1.1 ping statistics ---
302 packets transmitted, 9 packets received, 97% packet loss
round-trip min/avg/max/stddev = 2.816/450.054/2336.660/785.635 ms
```

You can check that sessions are now being created in the session flow table with the `show security flow session` command:

```
lab@vSRX-1> show security flow session protocol icmp
Session ID: 149, Policy name: HostB-to-HostA/4, Timeout: 2, Valid
  In: 200.1.1.20/46 --> 200.1.1.1/2722;icmp, Conn Tag: 0x0, If: ge-0/0/1.0, Pkts: 1, Bytes: 84,
  Out: 10.1.1.1/2722 --> 200.1.1.20/46;icmp, Conn Tag: 0x0, If: ge-0/0/2.0, Pkts: 1, Bytes: 84,

Session ID: 150, Policy name: HostB-to-HostA/4, Timeout: 4, Valid
  In: 200.1.1.20/47 --> 200.1.1.1/2722;icmp, Conn Tag: 0x0, If: ge-0/0/1.0, Pkts: 1, Bytes: 84,
  Out: 10.1.1.1/2722 --> 200.1.1.20/47;icmp, Conn Tag: 0x0, If: ge-0/0/2.0, Pkts: 1, Bytes: 84,
Total sessions: 2
```

You can see in the output that the destination address is being translated by comparing the addresses of the two entries that are part of the session, as shown in Figure 13.10.



```
lab@vSRX-1> show security flow session protocol icmp

Session ID: 149, Policy name: HostB-to-HostA/4, Timeout: 2, Valid

1 In: 200.1.1.20/46 --> 200.1.1.1/2722;icmp, Conn Tag: 0x0, If: ge-0/0/1.0, Pkts: 1, Bytes: 84,

2 Out: 10.1.1.1/2722 --> 200.1.1.20/46;icmp, Conn Tag: 0x0, If: ge-0/0/2.0, Pkts: 1, Bytes: 84,
```

Figure 13.10 Show Security Flow Session Output

You can also verify that the address is being translated to an address from the expected pool of addresses by looking at the destination NAT pool status and statistics:

```
lab@vSRX-1> clear security nat statistics destination pool DNAT-POOL-10

lab@vSRX-2> ping 200.1.1.1 routing-instance HOSTB count 10 rapid
PING 200.1.1.1 (200.1.1.1): 56 data bytes
!!!!!!!!!!
```

```
--- 200.1.1.1 ping statistics ---
10 packets transmitted, 10 packets received, 0% packet loss
round-trip min/avg/max/stddev = 2.547/8.060/18.905/4.458 ms

lab@vSRX-1> show security nat destination pool DNAT-POOL-10
Pool name       : DNAT-POOL-10
Pool id         : 1
Total address   : 1
Translation hits: 10
Address range                    Port
      10.1.1.1 - 10.1.1.1           0
```

Lastly, you can look at the log file that we configured under system syslog, to see the messages generated when the sessions are created and closed:

```
lab@vSRX-1> show log SECURITY
---more---
Mar  7 10:32:59  vSRX-1 RT_FLOW: RT_FLOW_SESSION_CREATE: session created 200.1.1.20/9->200.1.1.1/37870
0x0 icmp 200.1.1.20/9->10.1.1.1/37870 0x0 N/A N/A destination rule DNAT-200-to-10 1 HostB-to-HostA
UNTRUST TRUST 3082 N/A(N/A) ge-0/0/1.0 UNKNOWN UNKNOWN UNKNOWN
Mar  7 10:33:03  vSRX-1 RT_FLOW: RT_FLOW_SESSION_CLOSE: session closed response received:
200.1.1.20/0->200.1.1.1/37870 0x0 icmp 200.1.1.20/0->10.1.1.1/37870 0x0 N/A N/A destination rule
DNAT-200-to-10 1 HostB-to-HostA UNTRUST TRUST 3073 1(84) 1(84) 3 UNKNOWN UNKNOWN N/A(N/A) ge-0/0/1.0
UNKNOWN
```

IPv4 Source NAT and Proxy ARP

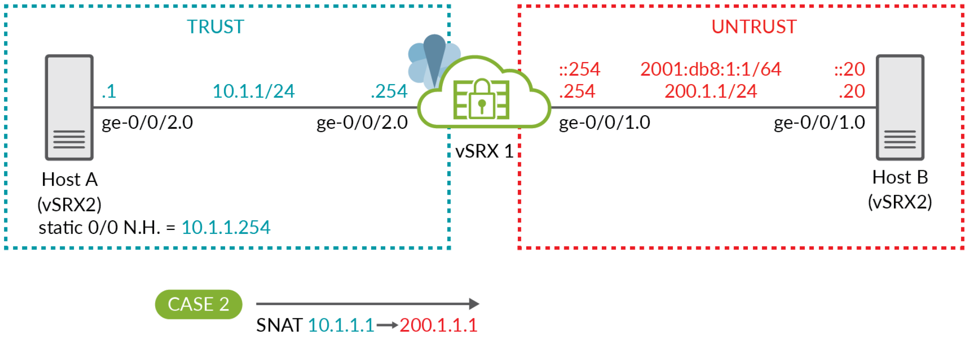


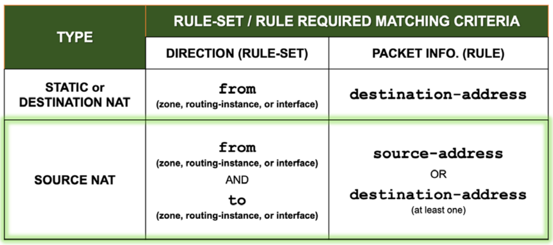
Figure 13.11 Case #2 – Source NAT

Create a `rule-set` for source NAT, and specify where the traffic to be translated is coming from and going to:

```
[edit security nat source]
lab@vSRX-1# show
rule-set from-trust-source-rule-set {
    from zone TRUST;
    to zone UNTRUST;
}
```

```
[edit security nat source]
lab@vSRX-1# show | display set relative
set rule-set from-trust-source-rule-set from zone TRUST
set rule-set from-trust-source-rule-set to zone UNTRUST
```

NOTE For source NAT, configuring the from and to statement, AND matching on either the `source-address` or the `destination-address` are mandatory. The following table summarizes the requirements.



TYPE	RULE-SET / RULE REQUIRED MATCHING CRITERIA	
	DIRECTION (RULE-SET)	PACKET INFO. (RULE)
STATIC or DESTINATION NAT	`from` (zone, routing-instance, or interface)	`destination-address`
SOURCE NAT	`from` (zone, routing-instance, or interface) AND `to` (zone, routing-instance, or interface)	`source-address` OR `destination-address` (at least one)

Create a translation rule within the rule set, that matches source address 10.1.1.1/32, and translates it into 200.1.1.1/32:

```
[edit security nat source rule-set from-trust-source-rule-set ]
lab@vSRX-1# show
from zone TRUST;
to zone UNTRUST;
rule SNAT-10-to-200 {
   match {
      source-address 10.1.1.1/32;
   }
   then {
      source-nat {
         pool {
            SNAT-POOL-200;
         }
      }
   }
}

[edit security nat source rule-set from-trust-source-rule-set ]
lab@vSRX-1# show | display set relative
set from zone TRUST
set to zone UNTRUST
set rule SNAT-10-to-200 match source-address 10.1.1.1/32
set rule SNAT-10-to-200 then source-nat pool SNAT-POOL-200

[edit security nat source pool SNAT-POOL-200]
lab@vSRX-1# show
address 200.1.1.1/32;
```

```
[edit security nat source pool SNAT-POOL-200]
lab@vSRX-1# show | display set
set security nat source pool SNAT-POOL-200 address 200.1.1.1/32
```

Configure a security policy that allows HostB to reach HostA using any application:

```
[edit security policies from-zone TRUST to-zone UNTRUST]
lab@vSRX-1# show policy HostA-to-HostB
   match {
      source-address HOSTA-10.1.1.1/32;
      destination-address HOSTB-200.1.1.20;
      application any;
   }
   then {
      permit;
      log {
         session-init;
         session-close;
      }
   }

[edit security policies from-zone TRUST to-zone UNTRUST]
lab@vSRX-1# show policy HostA-to-HostB | display set relative
set policy HostA-to-HostB match source-address HOSTA-10.1.1.1/32
set policy HostA-to-HostB match destination-address HOSTB-200.1.1.20
set policy HostA-to-HostB match application any
set policy HostA-to-HostB then permit
set policy HostA-to-HostB then log session-init
set policy HostA-to-HostB then log session-close
```

NOTE When writing the policy we used the untranslated source address of HostA.

Source NAT is applied *after* the policy is applied to the traffic, as shown in Figure 13.12. Therefore, you need to make sure to match on the untranslated source address.

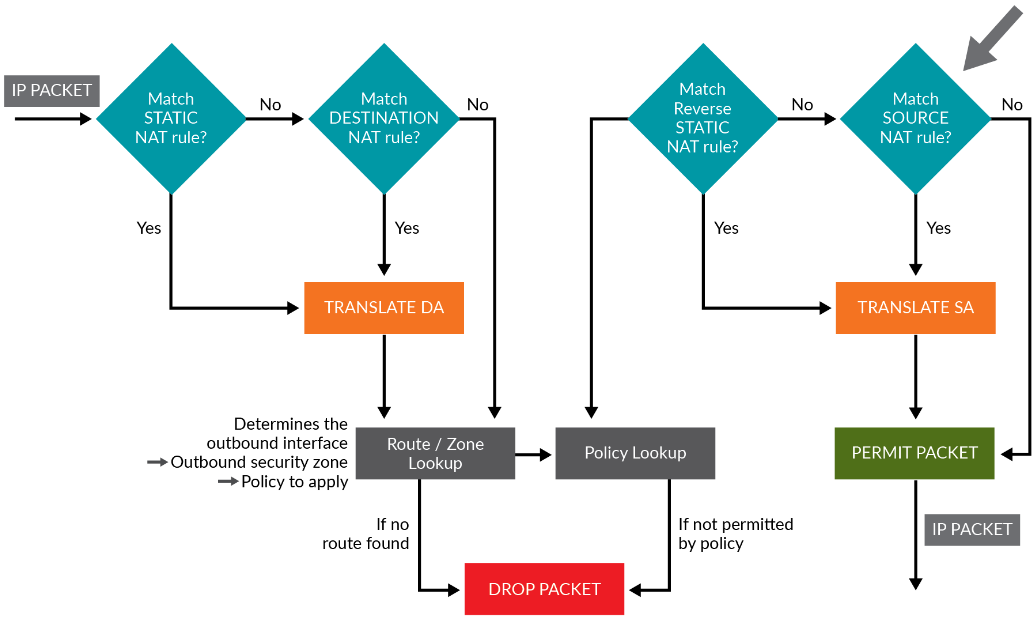


Figure 13.12 Packet Processing with NAT

Verify that HostA Can Communicate with HostB.

Because Proxy-ARP is already configured on ge-0/0/1 to respond to ARP requests for 200.1.1.1/32, we should see that HostA can reach HostB without any further configuration changes:

```
[edit security nat]
lab@vSRX-1# show | display set | match proxy
set security nat proxy-arp interface ge-0/0/1.0 address 200.1.1.1/32

lab@vSRX-2> ping 200.1.1.20 routing-instance HOSTA rapid
PING 200.1.1.20 (200.1.1.20): 56 data bytes
!!!!!
--- 200.1.1.20 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max/stddev = 4.136/15.929/46.786/16.061 ms

lab@vSRX-1> monitor traffic interface ge-0/0/1.0 detail
Address resolution is ON. Use <no-resolve> to avoid any reverse lookup delay.
Address resolution timeout is 4s.
Listening on ge-0/0/1.0, capture size 1514 bytes

Reverse lookup for 200.1.1.20 failed (check DNS reachability).
Other reverse lookup failures will not be reported.
Use <no-resolve> to avoid reverse lookups on IP addresses.
```

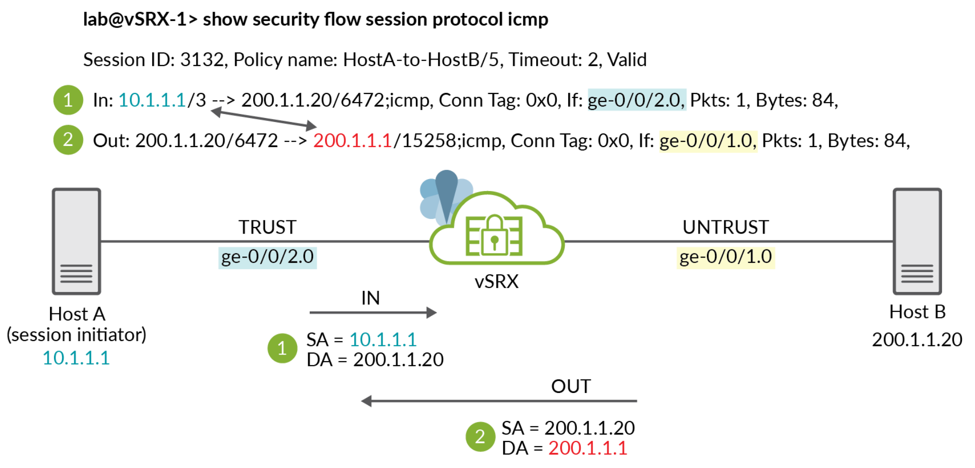
```
13:12:13.366558  In arp who-has 200.1.1.1 tell 200.1.1.20
13:12:13.367058 Out arp reply 200.1.1.1 is-at 2c:c2:60:5a:09:b2

lab@vSRX-2> show arp interface ge-0/0/1.0
MAC Address       Address         Name       Interface     Flags
2c:c2:60:5a:09:b2 200.1.1.1      200.1.1.1   ge-0/0/1.0    none
```

You can check that sessions are being created in the session flow table with the `show security flow session` command:

```
lab@vSRX-1> show security flow session protocol icmp
----more---
Session ID: 3132, Policy name: HostA-to-HostB/5, Timeout: 2, Valid
  In: 10.1.1.1/3 --> 200.1.1.20/6472;icmp, Conn Tag: 0x0, If: ge-0/0/2.0, Pkts: 1, Bytes: 84,
  Out: 200.1.1.20/6472 --> 200.1.1.1/15258;icmp, Conn Tag: 0x0, If: ge-0/0/1.0, Pkts: 1, Bytes: 84,
Session ID: 3133, Policy name: HostA-to-HostB/5, Timeout: 2, Valid
  In: 10.1.1.1/4 --> 200.1.1.20/6472;icmp, Conn Tag: 0x0, If: ge-0/0/2.0, Pkts: 1, Bytes: 84,
  Out: 200.1.1.20/6472 --> 200.1.1.1/13453;icmp, Conn Tag: 0x0, If: ge-0/0/1.0, Pkts: 1, Bytes: 84,
Total sessions: 5
```

You can see in the output that the source address is being translated, by comparing the addresses of the two entries that are part of the session, as shown in Figure 13.13.

You can also verify that the address is being translated to an address from the expected pool of addresses, by looking at the source NAT pool pool status and statistics.



```
lab@vSRX-1> show security flow session protocol icmp
Session ID: 3132, Policy name: HostA-to-HostB/5, Timeout: 2, Valid
1 In: 10.1.1.1/3 --> 200.1.1.20/6472;icmp, Conn Tag: 0x0, If: ge-0/0/2.0, Pkts: 1, Bytes: 84,
2 Out: 200.1.1.20/6472 --> 200.1.1.1/15258;icmp, Conn Tag: 0x0, If: ge-0/0/1.0, Pkts: 1, Bytes: 84,
```

Figure 13.13 Show Security Flow Session Output

```
lab@vSRX-1> clear security nat statistics source pool SNAT-POOL-200

lab@vSRX-2> ping 200.1.1.20 routing-instance HOSTA rapid
PING 200.1.1.20 (200.1.1.20): 56 data bytes
!!!!!
```

```
--- 200.1.1.20 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max/stddev = 2.602/9.089/23.701/7.782 ms

lab@vSRX-1> show security nat source pool SNAT-POOL-200
Pool name         : SNAT-POOL-200
Pool id           : 4
Routing instance   : default
Host address base  : 0.0.0.0
Port              : [1024, 63487]
Twin port         : [63488, 65535]
Port overloading   : 1
Address assignment : no-paired
Total addresses    : 1
Translation hits   : 5
Address range                   Single Ports   Twin Ports
        200.1.1.1 - 200.1.1.1        0            0
Total used ports   :                 0            0
```

Lastly, you can look at the log file that we configured under system syslog to see the messages generated when the sessions were created and closed:

```
lab@vSRX-1> show log SECURITY | find SNAT
Mar  7 13:30:24  vSRX-1 RT_FLOW: RT_FLOW_SESSION_CREATE: session created 10.1.1.1/0->200.1.1.20/25424
0x0 icmp 200.1.1.1/25351->200.1.1.20/25424 0x0 source rule SNAT-10-to-200 N/A N/A 1 HostA-to----
more---
Mar  7 13:30:26  vSRX-1 RT_FLOW: RT_FLOW_SESSION_CLOSE: session closed response received: 10.1.1.1/0-
>200.1.1.20/25424 0x0 icmp 200.1.1.1/25351->200.1.1.20/25424 0x0 source rule SNAT-10-to-200 N/A N/A 1
HostA-to-HostB TRUST UNTRUST 3139 1(84) 1(84) 2 UNKNOWN UNKNOWN N/A(N/A) ge-0/0/2.0 UNKNOWN
---more---
```

IPv6 to IPv4 Source and Destination NAT and Proxy-ARP/Proxy-NDP

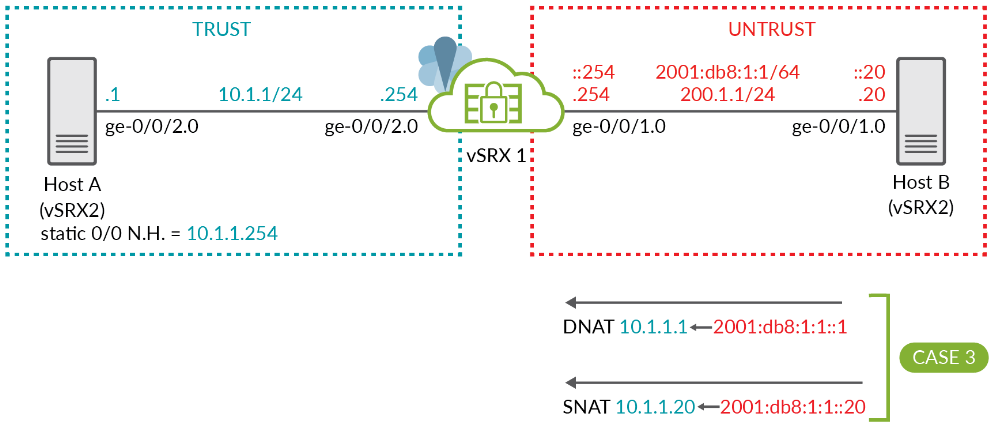


Figure 13.14 Case #3 – IPv6 to IPv4 Source and Destination NAT

Create a `rule-set` for Source NAT, and specify where the traffic to be translated is coming from and going to:

```
[edit security nat source]
lab@vSRX-1# show
rule-set from-untrust-source-rule-set {
   from zone UNTRUST;
   to zone TRUST;
}

[edit security nat source]
lab@vSRX-1# show | display set relative
set rule-set from-untrust-source-rule-set from zone UNTRUST
set rule-set from-untrust-source-rule-set to zone TRUST
```

NOTE We do not need to create a `destination rule-set` for this case, because we already have a rule set from zone UNTRUST.

Create translation rules, within the rule set, that:

- match destination address 2001:db8:1:1::1/128, and translate it into 10.1.1.1/32.
- match source address 2001:db8:1:1::20/128, and translate it into 10.1.1.20/32.

```
[edit security nat destination rule-set from-untrust-dest-rule-set ]
lab@vSRX-1# show
from zone UNTRUST;
rule DNAT-IPv6-to-IPv4 {
   match {
      destination-address 2001:db8:1:1::1/128;
      source-address 2001:db8:1:1::20/128;
   }
   then {
      destination-nat {
         pool {
            DNAT-POOL-10;
         }
      }
   }
}

[edit security nat destination rule-set from-untrust-dest-rule-set ]
lab@vSRX-1# show | display set relative
set from zone UNTRUST
set rule DNAT-IPv6-to-IPv4 match destination-address 2001:db8:1:1::1/128
set rule DNAT-IPv6-to-IPv4 match source-address 2001:db8:1:1::20/128
set rule DNAT-IPv6-to-IPv4 then destination-nat pool DNAT-POOL-10

[edit security nat destination pool DNAT-POOL-10]
lab@vSRX-1# show
address 10.1.1.1/32;

[edit security nat destination pool DNAT-POOL-10]
lab@vSRX-1# show | display set
set security nat destination pool DNAT-POOL-10 address 10.1.1.1/32
```

NOTE You can use the same destination address pool that you used for IPv4 destination NAT:

```
[edit security nat source rule-set from-untrust-source-rule-set ]
lab@vSRX-1# show
from zone TRUST;
to zone UNTRUST;
rule SNAT-IPv6-to-IPv4 {
   match {
      destination-address 10.1.1.1/32;
      source address 2001:db8:1::1::20/128;
   }
   then {
      destination-nat {
         pool {
            SNAT-POOL-10;
         }
      }
   }
}

[edit security nat destination rule-set from-untrust-dest-rule-set ]
lab@vSRX-1# show | display set relative
set from zone UNTRUST
set rule SNAT-IPv6-to-IPv4 match source-address 2001:db8:1:1::20/128
set rule SNAT-IPv6-to-IPv4 match destination-address 10.1.1.1/32
set rule SNAT-IPv6-to-IPv4 then source-nat pool SNAT-POOL-10
```

NOTE Even though the requirement for source NAT is to match on *either* the destination address or the source address, for IPv6/IPv4 translation, make sure you configure both the source and destination addresses as match criteria. If you don't configure the destination address you will see an error message similar to this:

```
  'pool'
    src-pool (SNAT-POOL-10) is ipv4, while dst addr is ipv6.
error: configuration check-out failed
```

Also, because source NAT is applied after destination NAT, you need to match on the IPv4 address of HostA (translated destination address of HostA):

```
[edit security nat source pool SNAT-POOL-10]
lab@vSRX-1# show
address 10.1.1.20/32;

[edit security nat source pool SNAT-POOL-10]
lab@vSRX-1# show | display set
set security nat source pool SNAT-POOL-10 address 10.1.1.20/32
```

Configure a security policy that allows HostB to reach HostA using any application, and using IPv6:

```
[edit security policies from-zone UNTRUST to-zone TRUST]
lab@vSRX-1# show policy HostB-to-HostA-IPv6
```

```
   match {
      source-address HOSTB-IPv6;
      destination-address HOSTA-10.1.1.1/32;
      application any;
   }
   then {
      permit;
      log {
         session-init;
         session-close;
      }
   }

[edit security policies from-zone UNTRUST to-zone TRUST]
lab@vSRX-1# show | display set relative
set policy HostB-to-HostA-IPv6 match source-address HOSTB-IPv6
set policy HostB-to-HostA-IPv6 match destination-address HOSTA-10.1.1.1/32
set policy HostB-to-HostA-IPv6 match application any
set policy HostB-to-HostA-IPv6 then permit
set policy HostB-to-HostA-IPv6 then log session-init
set policy HostB-to-HostA-IPv6 then log session-close
```

Verify that HostB can communicate with HostA.

Communication between HostB and HostA using IPv6 fails at this point:

```
lab@vSRX-2> ping 2001:db8:1:1::1 source 2001:db8:1:1::20 routing-instance HOSTB
PING6(56=40+8+8 bytes) 2001:db8:1:1::20 --> 2001:db8:1:1::1
..........
--- 2001:db8:1:1::1 ping6 statistics ---
10 packets transmitted, 0 packets received, 100% packet loss
```

If you check the creation of flow sessions, you'll find that none has been created:

```
lab@vSRX-1> show security flow session protocol icmp
Total sessions: 0
```

Also, if you monitor traffic on interface ge-0/0/1.0, on vSRX2, you can see that HostB is attempting to resolve the IPv6 addresses of HostA, using Neighbor Solicitation messages, but is not getting any responses:

```
lab@vSRX-2> monitor traffic interface ge-0/0/1.0
verbose output suppressed, use <detail> or <extensive> for full protocol decode
Address resolution is ON. Use <no-resolve> to avoid any reverse lookup delay.
Address resolution timeout is 4s.
Listening on ge-0/0/1.0, capture size 96 bytes

Reverse lookup for ff02::1:ff00:1 failed (check DNS reachability).
Other reverse lookup failures will not be reported.
Use <no-resolve> to avoid reverse lookups on IP addresses.

----more----
23:36:48.912815 Out IP6 truncated-ip6 - 12 bytes missing!2001:db8:1:1::20 > ff02::1:ff00:1: ICMP6,
neighbor solicitation[|icmp6]
23:36:49.639756 Out IP6 truncated-ip6 - 12 bytes missing!2001:db8:1:1::20 > ff02::1:ff00:1: ICMP6,
neighbor solicitation[|icmp6]
23:36:49.961305 Out IP6 truncated-ip6 - 12 bytes missing!2001:db8:1:1::20 > ff02::1:ff00:1: ICMP6,
neighbor solicitation[|icmp6]
^C
```

```
21 packets received by filter
0 packets dropped by kernel
```

Configure proxy-NDP on interface ge-0/0/1.0 in vSRX1 to respond to ARP requests for 2001:db8:1:1::1.

Add `proxy-NDP` under edit `security nat` and `commit` the configuration:

```
[edit security nat proxy-ndp]
lab@vSRX-1# show
interface ge-0/0/.0 {
   address {
      2001:db8:1:1::1/128;
   }
}

[edit security nat proxy-ndp]
lab@vSRX-1# show | display set
set security nat proxy-NDP interface ge-0/0/1.0 address 2001:db8:1:1::1/128

[edit security nat proxy-arp]
lab@vSRX-1# commit
commit complete
```

As soon as the new vSRX1 configuration is activated, we see that vSRX-1 sends a Neighbor Advertisement message (NA) to HostB, as a response to its Neighbor Solicitation. We also see that HostB starts sending ICMPv6 echo requests messages:

```
lab@vSRX-2> monitor traffic interface ge-0/0/1.0
verbose output suppressed, use <detail> or <extensive> for full protocol decode
Address resolution is ON. Use <no-resolve> to avoid any reverse lookup delay.
Address resolution timeout is 4s.
Listening on ge-0/0/1.0, capture size 96 bytes

Reverse lookup for ff02::1:ff00:1 failed (check DNS reachability).
Other reverse lookup failures will not be reported.
Use <no-resolve> to avoid reverse lookups on IP addresses.

----more----
23:41:30.085391 Out IP6 truncated-ip6 - 12 bytes missing!2001:db8:1:1::20 > ff02::1:ff00:1: ICMP6,
neighbor solicitation[|icmp6]
23:41:32.250926 Out IP6 truncated-ip6 - 12 bytes missing!2001:db8:1:1::20 > ff02::1:ff00:1: ICMP6,
neighbor solicitation[|icmp6]
23:41:33.268310 Out IP6 truncated-ip6 - 12 bytes missing!2001:db8:1:1::20 > ff02::1:ff00:1: ICMP6,
neighbor solicitation[|icmp6]
23:41:34.158803  In IP6 2001:db8:1:1::254 > 2001:db8:1:1::20: ICMP6, neighbor advertisement, tgt is
2001:db8:1:1::1, length 32
23:41:34.173683 Out IP6 2001:db8:1:1::20 > 2001:db8:1:1::1: ICMP6, echo request, seq 343, length 16
---more---
23:41:38.632278 Out IP6 2001:db8:1:1::20 > 2001:db8:1:1::1: ICMP6, echo request, seq 350, length 16
23:41:39.649327 Out IP6 2001:db8:1:1::20 > 2001:db8:1:1::1: ICMP6, echo request, seq 351, length 16
23:41:40.637043 Out IP6 2001:db8:1:1::20 > 2001:db8:1:1::1: ICMP6, echo request, seq 352, length 16
^C
29 packets received by filter
0 packets dropped by kernel
```

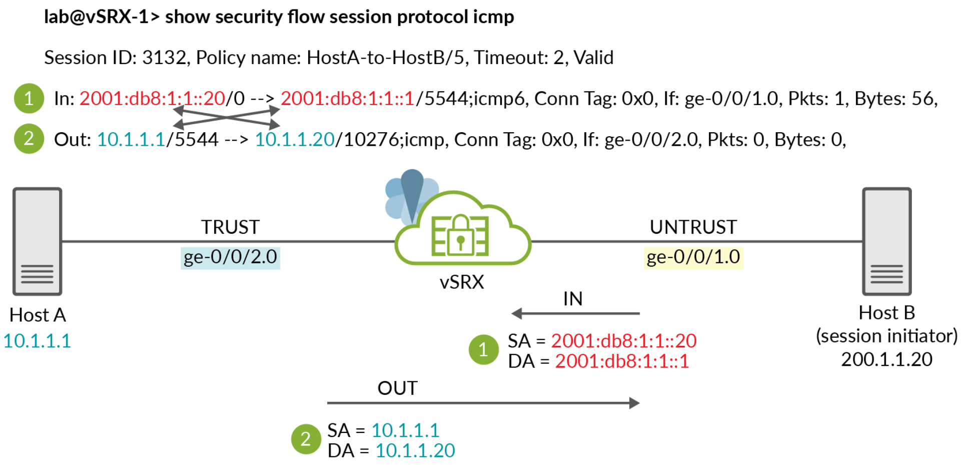
```
lab@vSRX-1> show interfaces ge-0/0/1 extensive | match hardw
  Current address: 2c:c2:60:5a:09:b2, Hardware address: 2c:c2:60:5a:09:b2

lab@vSRX-2> show ipv6 neighbors
IPv6 Address    Linklayer Address  State     Exp Rtr Secure interface
2001:db8:1:1::1 2c:c2:60:5a:09:b2  reachable 8   yes no    ge-0/0/1.0
```

vSRX1 is responding to the NS messages using its own MAC address. However, the ping is still not successful:

```
lab@vSRX-2> ping 2001:db8:1:1::1 source 2001:db8:1:1::20 routing-instance HOSTB
PING6(56=40+8+8 bytes) 2001:db8:1:1::20 --> 2001:db8:1:1::1
..........
--- 2001:db8:1:1::1 ping6 statistics ---
718 packets transmitted, 0 packets received, 100% packet loss
```

Check the creation of flow sessions, and that both the source address and the destination address are being translated, as expected:

```
lab@vSRX-1> show security flow session protocol icmp
Session ID: 3497, Policy name: HostB-to-HostA-IPv6/6, Timeout: 34, Valid
  In: 2001:db8:1:1::20/0 --> 2001:db8:1:1::1/5544;icmp6, Conn Tag: 0x0, If: ge-0/0/1.0, Pkts: 1, Bytes:
56,
  Out: 10.1.1.1/5544 --> 10.1.1.20/10276;icmp, Conn Tag: 0x0, If: ge-0/0/2.0, Pkts: 0, Bytes: 0,

Session ID: 3498, Policy name: HostB-to-HostA-IPv6/6, Timeout: 34, Valid
  In: 2001:db8:1:1::20/1 --> 2001:db8:1:1::1/5544;icmp6, Conn Tag: 0x0, If: ge-0/0/1.0, Pkts: 1, Bytes:
56,
  Out: 10.1.1.1/5544 --> 10.1.1.20/4752;icmp, Conn Tag: 0x0, If: ge-0/0/2.0, Pkts: 0, Bytes: 0,
Session ID: 3499, Policy name: HostB-to-HostA-IPv6/6, Timeout: 34, Valid
  In: 2001:db8:1:1::20/2 --> 2001:db8:1:1::1/5544;icmp6, Conn Tag: 0x0, If: ge-0/0/1.0, Pkts: 1, Bytes:
56,
  Out: 10.1.1.1/5544 --> 10.1.1.20/16688;icmp, Conn Tag: 0x0, If: ge-0/0/2.0, Pkts: 0, Bytes: 0,
---more-
```



```
lab@vSRX-1> show security flow session protocol icmp
Session ID: 3132, Policy name: HostA-to-HostB/5, Timeout: 2, Valid
1 In: 2001:db8:1:1::20/0 --> 2001:db8:1:1::1/5544;icmp6, Conn Tag: 0x0, If: ge-0/0/1.0, Pkts: 1, Bytes: 56,
2 Out: 10.1.1.1/5544 --> 10.1.1.20/10276;icmp, Conn Tag: 0x0, If: ge-0/0/2.0, Pkts: 0, Bytes: 0,
```

Figure 13.15 Show Security Flow Session Output

Check whether HostA is receiving the packets, and whether it is responding, by monitoring traffic on interface ge-0/0/2.0. You will find that HostA is attempting to resolve the SA address of the packets (10.1.1.20) to a MAC address, but is not successful.:

```
lab@vSRX-2> monitor traffic interface ge-0/0/2.0
verbose output suppressed, use <detail> or <extensive> for full protocol decode
Address resolution is ON. Use <no-resolve> to avoid any reverse lookup delay.
Address resolution timeout is 4s.
Listening on ge-0/0/2.0, capture size 96 bytes

Reverse lookup for 10.1.1.20 failed (check DNS reachability).
Other reverse lookup failures will not be reported.
Use <no-resolve> to avoid reverse lookups on IP addresses.

23:57:11.778389 Out arp who-has 10.1.1.20 tell 10.1.1.1
23:57:12.680536 Out arp who-has 10.1.1.20 tell 10.1.1.1
23:57:13.443351 Out arp who-has 10.1.1.20 tell 10.1.1.1
23:57:14.093624 Out arp who-has 10.1.1.20 tell 10.1.1.1
23:57:14.921522 Out arp who-has 10.1.1.20 tell 10.1.1.1
23:57:15.622606 Out arp who-has 10.1.1.20 tell 10.1.1.1
23:57:16.234023 Out arp who-has 10.1.1.20 tell 10.1.1.1
23:57:17.442487 Out arp who-has 10.1.1.20 tell 10.1.1.1
23:57:18.254331 Out arp who-has 10.1.1.20 tell 10.1.1.1
23:57:18.956331 Out arp who-has 10.1.1.20 tell 10.1.1.1
^C
11 packets received by filter
0 packets dropped by kernel
```

Configure proxy ARP on interface ge-0/0/2.0 in vSRX1 to respond to ARP requests for 10.1.1.20:

```
[edit security nat proxy-arp]
lab@vSRX-1# show
interface ge-0/0/1.0 {
   address {
      200.1.1.1/32;
   }
}
interface ge-0/0/2.0 {
   address {
      10.1.1.20/32;
   }
}

[edit security nat proxy-arp]
lab@vSRX-1# show | display set
set security nat proxy-arp interface ge-0/0/1.0 address 200.1.1.1/32
set security nat proxy-arp interface ge-0/0/2.0 address 10.1.1.20/32
```

NOTE More than one range of addresses can be configured for a given interface.

As soon as the new vSRX1 configuration is activated, we see that vSRX-1 sends an ARP reply to HostA, and traffic starts flowing from HostB to HostA:

```
lab@vSRX-2> monitor traffic interface ge-0/0/2.0
verbose output suppressed, use <detail> or <extensive> for full protocol decode
Address resolution is ON. Use <no-resolve> to avoid any reverse lookup delay.
Address resolution timeout is 4s.
Listening on ge-0/0/2.0, capture size 96 bytes

Reverse lookup for 10.1.1.20 failed (check DNS reachability).
Other reverse lookup failures will not be reported.
Use <no-resolve> to avoid reverse lookups on IP addresses.
---more---

00:08:22.937824 Out arp who-has 10.1.1.20 tell 10.1.1.1
00:08:23.548842 Out arp who-has 10.1.1.20 tell 10.1.1.1
00:08:24.159848 Out arp who-has 10.1.1.20 tell 10.1.1.1
00:08:25.449548 Out arp who-has 10.1.1.20 tell 10.1.1.1
00:08:26.180040 Out arp who-has 10.1.1.20 tell 10.1.1.1
00:08:26.796787 Out arp who-has 10.1.1.20 tell 10.1.1.1
00:08:27.398370 Out arp who-has 10.1.1.20 tell 10.1.1.1
00:08:28.325865 Out arp who-has 10.1.1.20 tell 10.1.1.1
00:08:29.451958 Out arp who-has 10.1.1.20 tell 10.1.1.1
00:08:29.455815  In arp reply 10.1.1.20 is-at 2c:c2:60:6d:55:0c
^C
28 packets received by filter
0 packets dropped by kernel
```

vSRX1 is responding to the ARP request using its own MAC address:

```
lab@vSRX-1> show interfaces ge-0/0/2 extensive | match hardw
  Current address: 2c:c2:60:6d:55:0c, Hardware address: 2c:c2:60:6d:55:0c

lab@vSRX-2> show arp interface ge-0/0/1.0
MAC Address      Address    Name       Interface    Flags
2c:c2:60:6d:55:0c 10.1.1.20   10.1.1.20    ge-0/0/2.0   none
2c:c2:60:6d:55:0c 10.1.1.254  10.1.1.254   ge-0/0/2.0   none
Total entries: 2

lab@vSRX-2> ping 2001:db8:1:1::1 source 2001:db8:1:1::20 routing-instance HOSTB
PING6(56=40+8+8 bytes) 2001:db8:1:1::20 --> 2001:db8:1:1::1
16 bytes from 2001:db8:1:1::1, icmp_seq=1086 hlim=63 time=84.395 ms
16 bytes from 2001:db8:1:1::1, icmp_seq=1087 hlim=63 time=6.308 ms
16 bytes from 2001:db8:1:1::1, icmp_seq=1088 hlim=63 time=3.695 ms
16 bytes from 2001:db8:1:1::1, icmp_seq=1089 hlim=63 time=3.206 ms
16 bytes from 2001:db8:1:1::1, icmp_seq=1090 hlim=63 time=3.785 ms
16 bytes from 2001:db8:1:1::1, icmp_seq=1091 hlim=63 time=5.130 ms
16 bytes from 2001:db8:1:1::1, icmp_seq=1092 hlim=63 time=2.884 ms
16 bytes from 2001:db8:1:1::1, icmp_seq=1093 hlim=63 time=2.948 ms
16 bytes from 2001:db8:1:1::1, icmp_seq=1094 hlim=63 time=4.777 ms
16 bytes from 2001:db8:1:1::1, icmp_seq=1095 hlim=63 time=3.470 ms
16 bytes from 2001:db8:1:1::1, icmp_seq=1096 hlim=63 time=3.030 ms
16 bytes from 2001:db8:1:1::1, icmp_seq=1097 hlim=63 time=5.890 ms
16 bytes from 2001:db8:1:1::1, icmp_seq=1098 hlim=63 time=8.097 ms
16 bytes from 2001:db8:1:1::1, icmp_seq=1099 hlim=63 time=3.532 ms
^C
--- 2001:db8:1:1::1 ping6 statistics ---
1100 packets transmitted, 14 packets received, 98% packet loss
round-trip min/avg/max/std-dev = 2.884/10.082/84.395/20.663 ms

lab@vSRX-1> show log SECURITY
---more---
```

```
Mar  7 21:18:43  vSRX-1 RT_FLOW: RT_FLOW_SESSION_CREATE: session created 2001:db8:1:1:0:0:0:20/241-
>2001:db8:1:1:0:0:0:1/9606 0x0 icmpv6 10.1.1.20/9440->10.1.1.1/9606 0x0 source rule SNAT-IPv6-to-IPv4 
destination rule DNAT-IPv6-to-IPv4 58 HostB-to-HostA-IPv6 UNTRUST TRUST 4753 N/A(N/A) ge-0/0/1.0 
UNKNOWN UNKNOWN UNKNOWN
Mar  7 21:18:44  vSRX-1 RT_FLOW: RT_FLOW_SESSION_CREATE: session created 2001:db8:1:1:0:0:0:20/242-
>2001:db8:1:1:0:0:0:1/9606 0x0 icmpv6 10.1.1.20/30081->10.1.1.1/9606 0x0 source rule SNAT-IPv6-to-
IPv4 destination rule DNAT-IPv6-to-IPv4 58 HostB-to-HostA-IPv6 UNTRUST TRUST 4754 N/A(N/A) ge-0/0/1.0 
UNKNOWN UNKNOWN UNKNOWN
Mar  7 21:18:45  vSRX-1 RT_FLOW: RT_FLOW_SESSION_CLOSE: session closed response received: 
2001:db8:1:1:0:0:0:20/239->2001:db8:1:1:0:0:0:1/9606 0x0 icmpv6 10.1.1.20/19023->10.1.1.1/9606 0x0 
source rule SNAT-IPv6-to-IPv4 destination rule DNAT-IPv6-to-IPv4 58 HostB-to-HostA-IPv6 UNTRUST TRUST 
4751 1(56) 1(36) 3 UNKNOWN UNKNOWN N/A(N/A) ge-0/0/1.0 UNKNOWN
---(more)---
```

Discussion

Proxy ARP and proxy NDP are *only* required when you are translating into an address in the same address range that your SRX interface, and the directly connected neighbors, belong to. You do not need to worry about proxy ARP/ND when you translate to the actual IP address configured on the SRX's interface, or when you are translating to an address in a different range. In the second case, you do need to make sure that remote devices have a route to the address range you are translating into.

For example, consider the scenario represented in Figure 13.16.

vSRX1 is translating the same address of HostA (10.1.1.1) to an address (211.1.1.1), that is not in the address range of its interface (200.1.1.0/24). Proxy ARP is *not* needed. However, HostB now needs to have a route to reach 211.1.1/24. This prefix is advertised by vSRX1 using BGP.

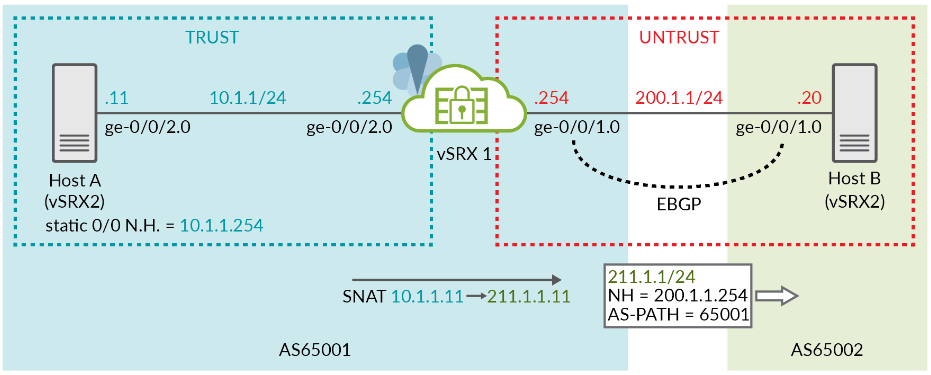


Figure 13.16 Source NAT Into an Address On a Different Address Range

The configuration statements for this case are shown below:

```
[edit security nat source rule-set from-trust-source-rule-set]
lab@vSRX-1# show
from zone TRUST;
to zone UNTRUST;
rule SNAT-10-to-211 {
   match {
      source-address 10.1.1.11/32;
   }
   then {
      source-nat {
         pool {
            SNAT-POOL-211;
         }
      }
   }
}

[edit security nat source pool SNAT-POOL-211]
lab@vSRX-1# show
address {
   211.1.1.11/32;
}

[edit security nat source pool SNAT-POOL-211]
lab@vSRX-1# show | display set relative
set address 211.1.1.11/32

[edit security nat source rule-set from-trust-source-rule-set]
lab@vSRX-1# show | display set relative
set from zone TRUST
set to zone UNTRUST
set rule SNAT-10-to-211 match source-address 10.1.1.11/32
set rule SNAT-10-to-211 then source-nat pool SNAT-POOL-211

[edit security address-book global]
lab@vSRX-1# show
address HOSTA-10.1.1.11/32 {
   description "Actual IPv4 address of HostA";
   10.1.1.11/32;
}
address HOSTA-211.1.1.1/32 {
   description "IPv4 address by which HostA is known by hosts in the UNTRUST Zone";
   211.1.1.11/32;
}

[edit security address-book global]
lab@vSRX-1# show | display set relative
set address HOSTA-10.1.1.11/32 description "Actual IPv4 address of HostA"
set address HOSTA-10.1.1.11/32 10.1.1.11/32
set address HOSTA-211.1.1.1/32 description "IPv4 address by which HostA is known by hosts in the
UNTRUST Zone"
set address HOSTA-211.1.1.1/32 211.1.1.11/32

[edit security policies from-zone TRUST to-zone UNTRUST policy HostA-to-HostB-211]
lab@vSRX-1# show
match {
   source-address HOSTA-10.1.1.11/32;
   destination-address HOSTB-200.1.1.20;
   application any;
}
then {
   permit;
   log {
      session-init;
      session-close;
   }
}

[edit security policies from-zone TRUST to-zone UNTRUST policy HostA-to-HostB-211]
lab@vSRX-1# show | display set relative
set match source-address HOSTA-10.1.1.11/32
set match destination-address HOSTB-200.1.1.20
set match application any
set then permit
set then log session-init
set then log session-close
```

At this point if you try to reach HostB from HostA using source address 10.1.1.11, you will see that a session is created, but connectivity fails:

```
[edit]
lab@vSRX-1# run show security flow session protocol icmp
Session ID: 16532, Policy name: HostA-to-HostB-211/7, Timeout: 44, Valid
  In: 10.1.1.11/0 --> 200.1.1.20/49265;icmp, Conn Tag: 0x0, If: ge-0/0/2.0, Pkts: 1, Bytes: 84,
  Out: 200.1.1.20/49265 --> 211.1.1.11/26984;icmp, Conn Tag: 0x0, If: ge-0/0/1.0, Pkts: 0, Bytes: 0,
```

You will also see packets coming in on ge-0/0/2 at HostB, but no replies are sent:

```
lab@vSRX-2> monitor interface ge-0/0/1
vSRX-2           Seconds: 19              Time: 10:32:42
                                         Delay: 7/2/33
Interface: ge-0/0/1.0, Enabled, Link is Up
Flags: SNMP-Traps 0x4004000
Encapsulation: ENET2
Local statistics:    Current delta
  Input bytes:      14992              [0]
  Output bytes:     19200             [234]
  Input packets:      241              [0]
  Output packets:     266              [3]
Remote statistics:
  Input bytes:      46704 (664 bps)    [1596]
  Output bytes:        0 (0 bps)        [0]
  Input packets:      556 (0 pps)       [19]
  Output packets:      0 (0 pps)        [0]
---more---
```

After configuring BGP and advertising a route for 211.1.1/24 to HostB, ping is successful:

```
[edit protocols bgp group EBGP]
lab@vSRX-1# show
type external;
export Prefix-211;
```

```
peer-as 65002;
local-as 65001;
neighbor 200.1.1.20;

[edit protocols bgp group EBGP]
lab@vSRX-1# show | display set relative
set type external
set export prefix-211
set peer-as 65002
set local-as 65001
set neighbor 200.1.1.20
deactivate protocols

[edit routing-options static]
lab@vSRX-1# show
set route 211.1.1/24 discard;

[edit routing-options static]
lab@vSRX-1# show | display set
set routing-options static route 211.1.1/24 discard

[edit policy-options policy-statement prefix-211]
lab@vSRX-1# show
term 1 {
   from {
      route-filter 211.1.1.0/24 exact;
   }
   then accept;
}

[edit policy-options policy-statement prefix-211]
lab@vSRX-1# show | display set relative
set term 1 from route-filter 211.1.1.0/24 exact
set term 1 then accept

lab@vSRX-2> ping 200.1.1.20 routing-instance HOSTA source 10.1.1.11 rapid
PING 200.1.1.20 (200.1.1.20): 56 data bytes
!!!!!
--- 200.1.1.20 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max/stddev = 3.366/5.768/9.824/2.185 ms
```

Recipe 14: Q-in-Q Tunneling Using ELS

By Paul Clarke

- Junos OS used: Junos 14.1X53-D47
- Juniper Platforms General Applicability: EX4600

Q-in-Q tunneling and VLAN translation allow service providers to create a Layer 2 Ethernet connection between two customer sites. Providers can segregate different customers' VLAN traffic on a link (for example, if the customers use overlapping VLAN IDs) or bundle different customer VLANs into a single-service VLAN. Data centers can use Q-in-Q tunneling and VLAN translation to isolate customer traffic within a single site, or to enable customer traffic flows between cloud data centers in different geographic locations.

Using Q-in-Q tunneling, providers can segregate or bundle customer traffic into fewer VLANs or different VLANs by adding another layer of 802.1Q tags. Q-in-Q tunneling is useful when customers have overlapping VLAN IDs, because the customer's 802.1Q (dot1Q) VLAN tags are prepended by the service VLAN (S-VLAN) tag. The Junos OS implementation of Q-in-Q tunneling supports the IEEE 802.1ad standard.

Problem

The evolution of the Junos OS and the introduction of ELS (Enhanced Layer 2 Software) means that the command line to configure Q-in-Q has changed.

Solution

With Q-in-Q tunneling, as a packet travels from a customer VLAN (C-VLAN) to a service provider's VLAN, a customer-specific 802.1Q tag is added to the packet. This additional tag is used to segregate traffic into service-provider-defined service VLANs (S-VLANs). The original customer 802.1Q tag of the packet remains and is transmitted transparently, passing through the service provider's network. As the packet leaves the S-VLAN in the downstream direction, the extra 802.1Q tag is removed.

There is nothing unusual about the configuration of the CE routers or the EX4300's in Figure 14.1. It's simply trunking VLANs from the router to the EX4300's and the EX4300's trunk VLANs towards the EX4600's. That's standard configuration.

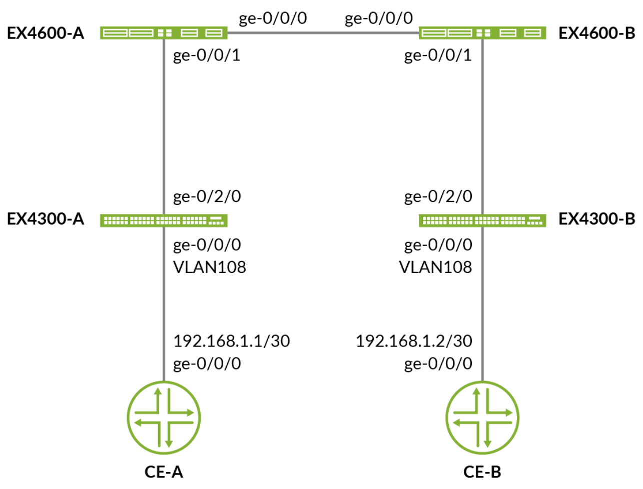


Figure 14.1 Recipe 14's Network Topology

For completeness, the configuration of the EX4300's and the SRX is shared here:

----- EX4300 Configuration -----

EX4300-A# show interfaces ge-0/0/0

```
unit 0 {
   family ethernet-switching {
      interface-mode trunk;
      vlan {
         members VLAN108;
      }
      storm-control default;
   }
}
```

EX4300-A# show interfaces ge-0/2/0

```
unit 0 {
   family ethernet-switching {
      interface-mode trunk;
      vlan {
         members VLAN108;
      }
      storm-control default;
   }
}
```

```
EX4300-A# show vlans
VLAN108 {
   vlan-id 108;
}
```

----- SRX CE Router Configuration -----

CE-A# show interfaces ge-0/0/0

```
vlan-tagging;
unit 108 {
   vlan-id 108;
   family inet {
      address 192.168.1.1/30;
   }
}
```

This recipe configures Q-in-Q tunneling using the *all-in-one bundling* method, which forwards all packets that ingress on a C-VLAN interface to an S-VLAN. (Packets are forwarded to the S-VLAN regardless of whether they are tagged or untagged prior to ingress.) Using this approach saves you the effort of specifying a specific mapping for each C-VLAN:

----- EX4600 inter Data Center configuration -----

EX4600-A# show interfaces ge-0/0/0

```
flexible-vlan-tagging;
encapsulation extended-vlan-bridge;
ether-options {
   ethernet-switch-profile {
      tag-protocol-id 0x8100;
   }
}
unit 95 {
   vlan-id 95;
}
```

So – what is the relevance of VLAN 95 and the other configuration items? Well, let's break it down item by item.

flexible-vlan-tagging

Support simultaneous transmission of 802.1Q VLAN single-tag and dual-tag frames on logical interfaces on the same Ethernet port, and on pseudowire logical interfaces.

encapsulation extended-vlan-bridge

Use extended VLAN bridge encapsulation on Ethernet interfaces that have IEEE 802.1Q VLAN tagging and bridging enabled and that must accept packets carrying TPID 0x8100 or a user-defined TPID:

```
ether-options {
   ethernet-switch-profile {
      tag-protocol-id 0x8100;
   }
}
```

Tag Protocol Identifier (TPID) is a 16-bit field set to a value of 0x8100 in order to identify the frame as an IEEE 802.1Q-tagged frame. This field is located at the same position as the EtherType field in untagged frames and is thus used to distinguish the frame from untagged frames.

VLAN 95 is configured to forward all packets that ingress on a C-VLAN interface to an S-VLAN. (Packets are forwarded to the S-VLAN regardless of whether they are tagged or untagged prior to ingress.) Using this approach saves you the effort of specifying a specific mapping for each C-VLAN:

EX4600-A# show vlans

```
VLAN95 {
   interface ge-0/0/0.95;
   interface ge-0/0/1.95;
}
```

NOTE Do not include the `vlan-id` in the VLAN configuration otherwise the configuration will not commit.

Now for the EX4600 interface facing the EX4300. Again, let's break it down item by item.

----- EX4600 downstream access switch configuration -----

EX4600-A# show interfaces ge-0/0/1

```
flexible-vlan-tagging;
native-vlan-id 150;
encapsulation extended-vlan-bridge;
unit 95 {
   vlan-id-list 100-200;
   input-vlan-map push;
   output-vlan-map pop;
}
```

flexible-vlan-tagging

Support simultaneous transmission of 802.1Q VLAN single-tag and dual-tag frames on logical interfaces on the same Ethernet port, and on pseudowire logical interfaces.

native-vlan-id 150

When the `native-vlan-id` statement is included with the `flexible-vlan-tagging` statement, untagged packets are accepted on the same mixed VLAN-tagged port and on the interfaces that are configured for Q-in-Q tunneling.

encapsulation extended-vlan-bridge

Use extended VLAN bridge encapsulation on Ethernet interfaces that have IEEE 802.1Q VLAN tagging and bridging enabled and that must accept packets carrying TPID 0x8100 or a user-defined TPID:

```
unit 95 {
   vlan-id-list 100-200;
   input-vlan-map push;
   output-vlan-map pop;
}
```

vlan-id-list

Specify a VLAN identifier list to use for a bridge domain or VLAN in trunk mode.

input-vlan-map push

Specify the VLAN rewrite operation to add a new VLAN tag to the top of the VLAN stack. An outer VLAN tag is pushed in front of the existing VLAN tag.

output-vlan-map pop

Specify the VLAN rewrite operation to remove a VLAN tag from the top of the VLAN tag stack. The outer VLAN tag of the frame is removed.

Okay, this solution is really neat because it means the customer can add VLANs as they please (within the specified range) and you don't need to make any changes on the EX4600 Data Center switches.

The configuration makes interface ge-0/0/1.95 a member of S-VLAN VLAN95, enables Q-in-Q tunneling, maps packets from C-VLANs 100 through 200 to S-VLAN 95, and enables ge-0/0/1 to accept untagged packets. If a packet originates in C-VLAN 108 and needs to be sent across the S-VLAN, a tag with VLAN ID 95 is added to the packet. When a packet is forwarded (internally) from the S-VLAN interface to interface ge-0/0/1, the tag with VLAN ID 95 is removed.

Let's test it. Below are the results of the ping tests that show ARP outputs. The focus is on VLAN 108. There's a second VLAN also configured using VLAN 116, which enters the EX4300 on a separate physical interface, hence it's not in the above configuration example:

CE-A# run show arp

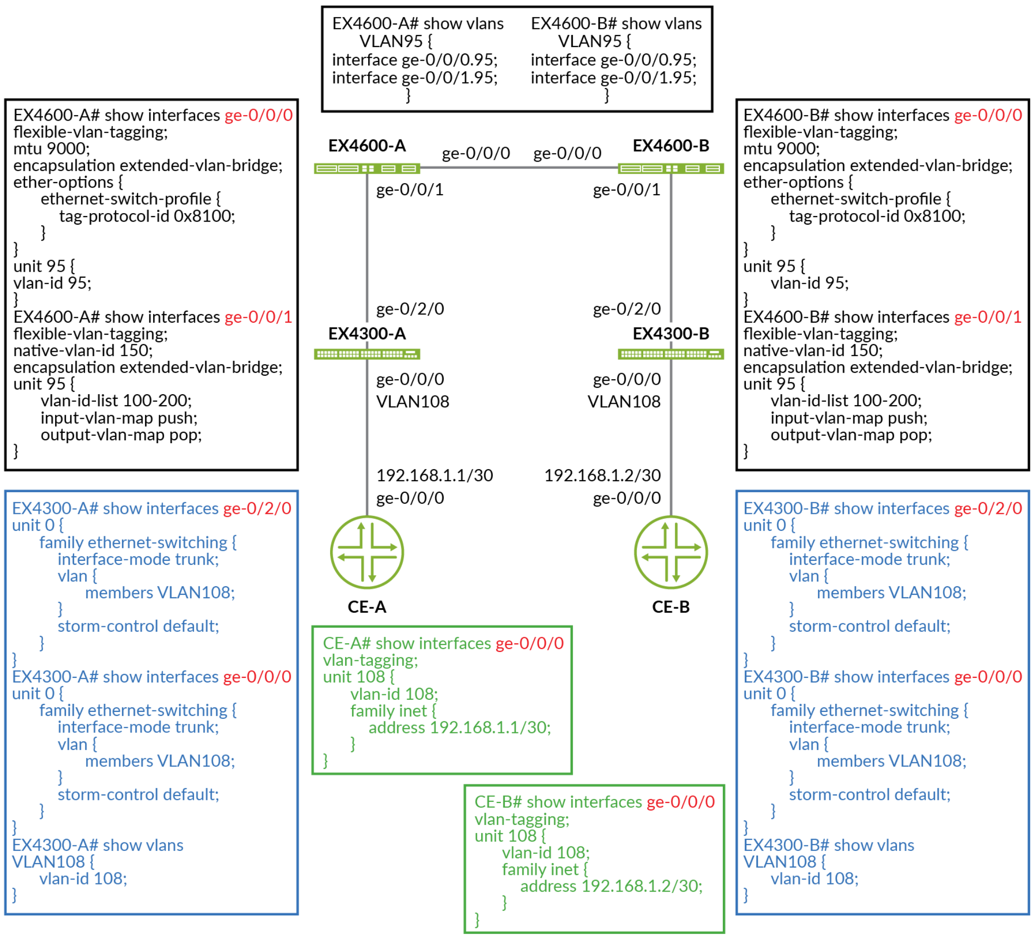
```
MAC Address              Address          Name                       Interface              Flags
00:31:46:9d:43:80      192.168.1.2     192.168.1.2             ge-0/0/0.108       none
00:31:46:9d:43:81      192.168.1.6     192.168.1.6             ge-0/0/1.116       none
Total entries: 2
```

CE-A# run ping 192.168.1.2 rapid

```
PING 192.168.1.2 (192.168.1.2): 56 data bytes
!!!!!
```

```
--- 192.168.1.2 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.570/0.625/0.742/0.068 ms
```

Finally, one of the things we like to do is produce a "solution on a page" that we can refer back to at any time. Figure 14.2 is the complete topology and configuration in a single illustration.



```
EX4600-A# show vlans
    VLAN95 {
interface ge-0/0/0.95;
interface ge-0/0/1.95;
         }
```

```
EX4600-B# show vlans
    VLAN95 {
interface ge-0/0/0.95;
interface ge-0/0/1.95;
         }
```

```
EX4600-A# show interfaces ge-0/0/0
flexible-vlan-tagging;
mtu 9000;
encapsulation extended-vlan-bridge;
ether-options {
     ethernet-switch-profile {
         tag-protocol-id 0x8100;
     }
}
unit 95 {
vlan-id 95;
}
EX4600-A# show interfaces ge-0/0/1
flexible-vlan-tagging;
native-vlan-id 150;
encapsulation extended-vlan-bridge;
unit 95 {
     vlan-id-list 100-200;
     input-vlan-map push;
     output-vlan-map pop;
}
```

```
EX4600-B# show interfaces ge-0/0/0
flexible-vlan-tagging;
mtu 9000;
encapsulation extended-vlan-bridge;
ether-options {
     ethernet-switch-profile {
         tag-protocol-id 0x8100;
     }
}
unit 95 {
     vlan-id 95;
}
EX4600-B# show interfaces ge-0/0/1
flexible-vlan-tagging;
native-vlan-id 150;
encapsulation extended-vlan-bridge;
unit 95 {
     vlan-id-list 100-200;
     input-vlan-map push;
     output-vlan-map pop;
}
```

```
EX4300-A# show interfaces ge-0/2/0
unit 0 {
     family ethernet-switching {
         interface-mode trunk;
         vlan {
               members VLAN108;
         }
         storm-control default;
     }
}
EX4300-A# show interfaces ge-0/0/0
unit 0 {
     family ethernet-switching {
         interface-mode trunk;
         vlan {
               members VLAN108;
         }
         storm-control default;
     }
}
EX4300-A# show vlans
VLAN108 {
     vlan-id 108;
}
```

```
CE-A# show interfaces ge-0/0/0
vlan-tagging;
unit 108 {
     vlan-id 108;
     family inet {
         address 192.168.1.1/30;
     }
}
```

```
CE-B# show interfaces ge-0/0/0
vlan-tagging;
unit 108 {
     vlan-id 108;
     family inet {
         address 192.168.1.2/30;
     }
}
```

```
EX4300-B# show interfaces ge-0/2/0
unit 0 {
     family ethernet-switching {
         interface-mode trunk;
         vlan {
               members VLAN108;
         }
         storm-control default;
     }
}
EX4300-B# show interfaces ge-0/0/0
unit 0 {
     family ethernet-switching {
         interface-mode trunk;
         vlan {
               members VLAN108;
         }
         storm-control default;
     }
}
EX4300-B# show vlans
VLAN108 {
     vlan-id 108;
}
```

Figure 14.2 Recipe 14's Solution On a Single Page

Discussion

When Q-in-Q tunneling is configured on EX Series switches, trunk interfaces are assumed to be part of the service-provider network and access interfaces are assumed to be part of the customer network. Therefore, this topic also refers to trunk interfaces as S-VLAN interfaces (network-to-network [NNI] interfaces, and to access interfaces as C-VLAN interfaces (user-to-network [UNI] interfaces).

This recipe has covered configuring Q-in-Q using all-in-one bundling. The Juniper TechLibrary documents other options to configure Q-in-Q tunneling by using one of the following methods to map C-VLANs to S-VLANs:

- Configuring All-in-One Bundling
- Configuring Many-to-Many Bundling
- Configuring a Specific Interface Mapping with VLAN Rewrite Option

Recipe 15: Low-Risk Methodology for Deploying Firewall Filters

By Stefan Fount

Juniper Platforms General Applicability: All routing, switching, and firewall platforms

Customers often want to further protect their networks by deploying stateless firewall filters on their Juniper routers, switches, and other Junos-based devices on their network. Even in the event that stateful firewalls are deployed, a properly designed network should encompass some level of stateless firewall filters so as to ensure that unnecessary traffic doesn't starve precious session table resources on stateful firewalls.

While there are many books and articles that focus on best practices for securing the control plane and the data plane in Junos using firewall filter constructs, and many more that talk about how to properly configure firewall filters, there aren't many that reference a simple and safe methodology for deploying them in a manner that essentially guarantees zero downtime in an environment.

This recipe attempts to introduce the reader to a low-risk methodology for safely deploying firewall filters, whether to filter data plane or control plane traffic, while maximizing uptime and reducing the likelihood of unintended consequences.

Problem

Network engineers traditionally look at the deployment of firewall filters on their networking equipment with great trepidation, especially on production systems where issues with deployment might arise or production traffic might be impacted. This typically results in deployment taking place during "maintenance windows," and in many cases takes multiple windows to get a fully fleshed firewall filter configuration in place.

Further exacerbating the problem is that network engineers might not realize the multitude of data plane and control plane traffic that is encompassed on their network, and therefore any initial approach at deploying firewall filters might be incomplete and not allow appropriate traffic through. The obvious result is that necessary protocols and services are effected, which necessitates turning off the firewall filters, leaving the environment more exposed than it should be.

Solution

There is actually a foolproof methodology for introducing firewall filters on Junos-based devices in a manner that is low risk and virtually guarantees zero downtime. It's so simple to deploy using this methodology that often times firewall filters can be deployed during normal business hours, which affords the network engineer ample time to ensure all traffic is accounted for.

The crux of this solution is to build an initial firewall filter encompassing all known protocols and services, and to use a final term which accepts all traffic with logging enabled, such that anything not accounted for is revealed through logs. Once everything in the environment has been accounted for, we can change the final term to a discard or a reject action with confidence that we have safely captured everything. This methodology could be extended over a few hours or even days to ensure everything is fully accounted for.

NOTE This cookbook recipe does not attempt to recommend firewall best practices or illustrate best approaches for building a structured firewall filter. There are many excellent references for the astute reader that cover these topics in more detail, however, one that is strongly recommended is *Day One: Securing the Routing Engine on M, MX, and T Series*, by Douglas Hanks Jr.

Let's start with a typical border router, with no firewall filters applied, and demonstrate this methodology in action:

```
root@Border-Router# show interfaces
ge-0/0/0 {
    unit 0 {
        family inet {
            address 33.0.0.2/30;
        }
    }
}
ge-0/0/1 {
    unit 0 {
        family inet {
            address 10.0.0.1/30;
        }
    }
}
fxp0 {
    unit 0 {
        family inet {
            address 172.16.110.151/24;
        }
    }
}
lo0 {
    unit 0 {
        family inet {
            address 192.168.1.1/32;
        }
    }
}
```

And you can see that there are currently no firewall filters applied on any of our interfaces.

Let's assume we want to apply a firewall filter on ingress traffic on our WAN link from our ISP, which is ge-0/0/0.0. We start by building a firewall filter with a single term, with logging and counting enabled, and commit our configuration. Initially, we'll use an `accept` action in the firewall filter to ensure that there is no impact to any production traffic in the environment. This will be changed to a `discard` action at the very end of this methodology:

```
=[edit]
root@Border-Router# show interfaces ge-0/0/0
unit 0 {
    family inet {
        filter {
            input ingress-filter;
        }
        address 33.0.0.2/30;
    }
}

[edit]
root@Border-Router# show firewall family inet filter ingress-filter
term final-discard {
    then {
        count final-discard;
        log;
        accept;
    }
}

[edit]
root@Border-Router# commit
commit complete
```

Once our firewall filter is applied, it's just a matter of identifying protocols in use on our network by simply looking at the firewall counters and the firewall log:

```
[edit]
root@Border-Router# run show firewall

Filter: __default_bpdu_filter__

Filter: ingress-filter
Counters:
Name                                                 Bytes             Packets
final-discard                                         1027                  14
```

You can clearly see that the `final-discard` term in our `ingress-filter` is matching traffic, so let's dig a little deeper by looking into the logs:

```
[edit]
root@Border-Router# run show firewall log detail
Time of Log: 2019-03-05 19:00:50 UTC, Filter: pfe, Filter action: accept, Name of interface: ge-0/0/0.0
Name of protocol: UDP, Packet Length: 13312, Source address: 33.0.0.1:49152, Destination address:
33.0.0.2:3784
Time of Log: 2019-03-05 19:00:46 UTC, Filter: pfe, Filter action: accept, Name of interface: ge-0/0/0.0
Name of protocol: ICMP, Packet Length: 20480, Source address: 33.0.0.1, Destination address: 33.0.0.2
ICMP type: 14, ICMP code: 0
Time of Log: 2019-03-05 19:00:45 UTC, Filter: pfe, Filter action: accept, Name of interface: ge-0/0/0.0
Name of protocol: ICMP, Packet Length: 20480, Source address: 33.0.0.1, Destination address: 33.0.0.2
ICMP type: 13, ICMP code: 0
Time of Log: 2019-03-05 19:00:42 UTC, Filter: pfe, Filter action: accept, Name of interface: ge-0/0/0.0
Name of protocol: UDP, Packet Length: 13312, Source address: 33.0.0.1:49152, Destination address:
33.0.0.2:3784
[Output truncated for brevity...]
```

Based on the output above, you can see at least two protocols in use, for example, ICMP, and BFD (as indicated via UDP port 3784). Let's include those in our firewall filter and add counters to the terms to ensure that those terms are now being matched properly. In addition, you need to ensure you insert the terms before the final catch-all `final-discard` term:

```
[edit firewall family inet filter ingress-filter]
root@Border-Router# set term icmp from protocol icmp

[edit firewall family inet filter ingress-filter]
root@Border-Router# set term icmp then count icmp

[edit firewall family inet filter ingress-filter]
root@Border-Router# set term icmp then accept

[edit firewall family inet filter ingress-filter]
root@Border-Router# insert term icmp before term final-discard

[edit firewall family inet filter ingress-filter]
root@Border-Router# set term bfd from protocol udp

[edit firewall family inet filter ingress-filter]
root@Border-Router# set term bfd from destination-port 3784-3785

[edit firewall family inet filter ingress-filter]
root@Border-Router# set term bfd then count bfd

[edit firewall family inet filter ingress-filter]
root@Border-Router# set term bfd then accept

[edit firewall family inet filter ingress-filter]
root@Border-Router# insert term bfd before term final-discard
```

As a result, the firewall filter configuration now looks like this:

```
[edit firewall family inet filter ingress-filter]
root@Border-Router# show
term icmp {
    from {
        protocol icmp;
    }
    then {
        count icmp;
        accept;
    }
}
term bfd {
    from {
        protocol udp;
        destination-port 3784-3785;
    }
    then {
        count bfd;
        accept;
    }
}
term final-discard {
    then {
        count final-discard;
        log;
        accept;
    }
}
```

After we commit, let's take a look a look at our counters and firewall logs to see if we have fully captured all the protocols that exist on our network that need to be accounted for. Beforehand, it's a good idea to clear the counters and the logs so that you can start from a clean slate:

```
[edit firewall family inet filter ingress-filter]
root@Border-Router# run clear firewall all

[edit firewall family inet filter ingress-filter]
root@Border-Router# run clear firewall log

[edit firewall family inet filter ingress-filter]
root@Border-Router# run show firewall

Filter: __default_bpdu_filter__

Filter: ingress-filter
Counters:
Name                                                Bytes              Packets
bfd                                                   312                    6
final-discard                                           0                    0
icmp                                                 1840                   23
```

At first glance, it does not appear that anything is matching our `final-discard` term and associated counter, but some protocols or services are not very chatty, so to be sure you should evaluate the counter over a period of time. The `refresh` knob, coupled with a match statement, is perfectly suited to see if anything appears over time.

TIP The `refresh` knob issues a continuous display of the command based on the interval specified in seconds. It proves to be very useful in a situation where the output might change over time:

```
[edit firewall family inet filter ingress-filter]
root@Border-Router# run show firewall | match final-discard | refresh 5
---(refreshed at 2019-03-05 19:25:11 UTC)---
final-discard                                  0                 0
---(refreshed at 2019-03-05 19:25:16 UTC)---
final-discard                                  0                 0
---(refreshed at 2019-03-05 19:25:21 UTC)---
final-discard                                  0                 0
---(refreshed at 2019-03-05 19:25:26 UTC)---
final-discard                                 52                 1
---(refreshed at 2019-03-05 19:25:31 UTC)---
final-discard                                 52                 1
---(refreshed at 2019-03-05 19:25:36 UTC)---
final-discard                                123                 2
---(*more 100%)---[abort]
```

Although it took a few seconds, as you can see from the output, the `final-discard` term continues to be matched against, which is a clear indicator that there is still some traffic that needs to be accounted for. Let's take a look at the log for further analysis:

```
[edit firewall family inet filter ingress-filter]
root@Border-Router# run show firewall log detail
Time of Log: 2019-03-05 19:25:32 UTC, Filter: pfe, Filter action: accept, Name of interface: ge-0/0/0.0
Name of protocol: TCP, Packet Length: 18176, Source address: 33.0.0.1:56016, Destination address:
33.0.0.2:179
Time of Log: 2019-03-05 19:25:22 UTC, Filter: pfe, Filter action: accept, Name of interface: ge-0/0/0.0
Name of protocol: TCP, Packet Length: 13312, Source address: 33.0.0.1:56016, Destination address:
33.0.0.2:179
```

In this case, you can see that BGP still needs to be accounted for (as indicated via TCP port 179). Let's add that to our firewall filter, and repeat the steps to ensure there is nothing left:

```
[edit firewall family inet filter ingress-filter]
root@Border-Router# set term bgp from protocol tcp destination-port bgp

[edit firewall family inet filter ingress-filter]
root@Border-Router# set term bgp then accept

[edit firewall family inet filter ingress-filter]
root@Border-Router# set term bgp then count bgp

[edit firewall family inet filter ingress-filter]
root@Border-Router# insert term bgp before term final-discard

[edit firewall family inet filter ingress-filter]
root@Border-Router# show
term icmp {
    from {
        protocol icmp;
    }
    then {
        count icmp;
        accept;
    }
}
term bfd {
    from {
        protocol udp;
        destination-port 3784-3785;
    }
    then {
        count bfd;
        accept;
    }
}
term bgp {
    from {
        protocol tcp;
        destination-port bgp;
    }
    then {
        count bgp;
        accept;
    }
}
term final-discard {
    then {
        count final-discard;
        log;
        accept;
    }
}

[edit firewall family inet filter ingress-filter]
root@Border-Router# commit
commit complete
```

Now that we've added a term for BGP and committed the configuration, let's evaluate our counters and firewall logs once again to determine if there is anything remaining that hasn't been accounted for:

```
[edit firewall family inet filter ingress-filter]
root@Border-Router# run clear firewall all

[edit firewall family inet filter ingress-filter]
root@Border-Router# run clear firewall log

[edit firewall family inet filter ingress-filter]
root@Border-Router# run show firewall

Filter: __default_bpdu_filter__

Filter: ingress-filter
```

```
Counters:
Name                                                Bytes              Packets
bfd                                                   156                    3
bgp                                                    60                    1
final-discard                                           0                    0
icmp                                                  720                    9

[edit firewall family inet filter ingress-filter]
root@Border-Router# run show firewall | match final-discard | refresh 5
---(refreshed at 2019-03-05 21:39:45 UTC)---
final-discard                                           0                    0
---(refreshed at 2019-03-05 21:39:50 UTC)---
final-discard                                           0                    0
---(refreshed at 2019-03-05 21:39:55 UTC)---
final-discard                                           0                    0
---(refreshed at 2019-03-05 21:40:00 UTC)---
final-discard                                         116                    2
---(refreshed at 2019-03-05 21:40:05 UTC)---
final-discard                                         116                    2
---(refreshed at 2019-03-05 21:40:10 UTC)---
final-discard                                         116                    2
---(*more 100%)---[abort]

[edit firewall family inet filter ingress-filter]
root@Border-Router#

[edit firewall family inet filter ingress-filter]
root@Border-Router# run show firewall log detail
Time of Log: 2019-03-05 21:39:58 UTC, Filter: pfe, Filter action: accept, Name of interface: ge-0/0/0.0
Name of protocol: TCP, Packet Length: 13312, Source address: 33.0.0.1:55893, Destination address:
33.0.0.2:639
Time of Log: 2019-03-05 21:39:58 UTC, Filter: pfe, Filter action: accept, Name of interface: ge-0/0/0.0
Name of protocol: TCP, Packet Length: 16384, Source address: 33.0.0.1:55893, Destination address:
33.0.0.2:639
```

In this case, you can see that MSDP still needs to be accounted for (as indicated via TCP port 639). Let's add that to our firewall filter, and repeat the steps once again:

```
[edit firewall family inet filter ingress-filter]
root@Border-Router# set term msdp from protocol tcp destination-port msdp

[edit firewall family inet filter ingress-filter]
root@Border-Router# set term msdp then accept

[edit firewall family inet filter ingress-filter]
root@Border-Router# set term msdp then count msdp

[edit firewall family inet filter ingress-filter]
root@Border-Router# insert term msdp before term final-discard

[edit firewall family inet filter ingress-filter]
root@Border-Router# show
term icmp {
    from {
        protocol icmp;
    }
    then {
        count icmp;
        accept;
    }
}
term bfd {
    from {
        protocol udp;
        destination-port 3784-3785;
    }
    then {
        count bfd;
        accept;
    }
}
term bgp {
    from {
        protocol tcp;
        destination-port bgp;
    }
    then {
        count bgp;
        accept;
    }
}
term msdp {
    from {
        protocol tcp;
        destination-port msdp;
    }
    then {
        count msdp;
        accept;
    }
}
term final-discard {
    then {
        count final-discard;
        log;
        accept;
    }
}

[edit firewall family inet filter ingress-filter]
root@Border-Router# commit
commit complete
```

Now that we've added a term for MSDP and committed the configuration, let's evaluate our counters and firewall logs once again to determine if there is anything remaining that hasn't been accounted for:

```
[edit firewall family inet filter ingress-filter]
root@Border-Router# run clear firewall all

[edit firewall family inet filter ingress-filter]
root@Border-Router# run clear firewall log
```

```
[edit firewall family inet filter ingress-filter]
root@Border-Router# run show firewall
Filter: __default_bpdu_filter__

Filter: ingress-filter
Counters:
Name                                                Bytes              Packets
bfd                                                  1508                   29
bgp                                                   246                    4
final-discard                                           0                    0
icmp                                                 8160                  102
msdp                                                   52                    1
```

Initially, nothing appears to be matching our `final-discard` counter based on the output above. But in order to be sure, let the filters soak for a period of time – several minutes at a minimum – but the longer you can let them soak, the better.

After several hours, evaluate the firewall:

```
root@Border-Router# run show firewall

Filter: __default_bpdu_filter__

Filter: ingress-filter
Counters:
Name                                                Bytes              Packets
bfd                                                 90064                 1732
bgp                                                 13899                  226
final-discard                                           0                    0
icmp                                               482720                 6034
msdp                                                 5564                  104
```

Based on this output, it seems we no longer have traffic matching our `final-discard` term, which is a pretty good indicator that all traffic is now being properly matched in respective terms.

Once you feel confident that you have encompassed all necessary protocols and services in your firewall filters, the final step in this methodology is to modify the `final-discard` term in your firewall filter from an `accept` action to a `discard` or `reject` action. This will ensure that protocols and services not intended for your environment will be dropped upon receipt. And you can periodically investigate firewall counters and logs to evaluate over time and ensure the filter remains up-to-date as a result:

```
[edit firewall family inet filter ingress-filter]
root@Border-Router# set term final-discard then discard

[edit firewall family inet filter ingress-filter]
root@Border-Router# show
term icmp {
    from {
        protocol icmp;
    }
    then {
        count icmp;
        accept;
    }
}
term bfd {
    from {
        protocol udp;
        destination-port 3784-3785;
    }
    then {
        count bfd;
        accept;
    }
}
term bgp {
    from {
        protocol tcp;
        destination-port bgp;
    }
    then {
        count bgp;
        accept;
    }
}
term msdp {
    from {
        protocol tcp;
        destination-port msdp;
    }
    then {
        count msdp;
        accept;
    }
}
term final-discard {
    then {
        count final-discard;
        log;
        discard;
    }
}

[edit firewall family inet filter ingress-filter]
root@Border-Router# commit
commit complete
```

Discussion

This methodology should prove useful for network engineers responsible for deploying initial firewall filters on Junos-based devices in a manner that virtually guarantees no down time, and affords the network engineer a certain level of confidence that the majority of necessary services and protocols have been accounted for. Using techniques such as those outlined above, firewall filters could potentially even be deployed during normal business hours outside traditional maintenance windows.

Recipe 16: Translating RSVP-signaled LSPs for Quick Troubleshooting Using PyEZ

By Said van de Klundert

Python Version Used: 2.7.14

xxxx Version Used: 1.3.1

Junos OS Used: 16.1R3-S8

Juniper Platforms General Applicability: (v)MX

Troubleshooting large scale networks that switch traffic along RSVP signaled lable-switched paths (LSPs) can be quite bothersome. Imagine that reports on packet-loss are coming in from various users in different parts of the network, or users are reporting a sudden increase in latency getting from A to B.

You will have to log in to the PE routers, grab the LSP information, and then use the Explicit Route Object (ERO) to figure out what path the LSP is taking through the network. After grabbing the ERO, you log into every node along the LSP and have to do another route lookup to figure out what the next interface/node is. It takes quite some time to build a list of nodes and interfaces that comprise the LSP.

This recipe does just that for you. You simply run a script and that script will inform you of exactly which nodes and interfaces are used by a certain LSP.

Problem

The main problem to quickly troubleshoot RSVP signaled LSPs in a large network is translating the LSP to the nodes and interfaces that the LSP traverses.

Solution

The solution is to script the translation of the LSPs. This is done by first gathering all the interface IP address information and storing it for future use. This information might already be available somewhere in a database or DNS server. This example simply gathers the information and stores it in a Python dictionary that you save to disk using a module called ‘pickle’.

After storing the IP interface and hostnames to a dictionary, we can start translating an LSP. We do that by logging in to a router, grabbing the ERO list of the LSP, and doing an IP lookup for every IP address in that list.

For this recipe, we are going to assume that the interior gateway protocol (IGP) in use is OSPF and that the core links are configured with /31s.

Gathering All the Interface IP Address Information and Storing It for Future Use

The script that we use to gather the information is the following:

```
# !/usr/bin/env python
#
#
# Created by Said van de Klundert
#
#
import pickle
from jnpr.junos import Device
import os, getpass

list_of_devices = []


username = os.getlogin(); pwd = getpass.getpass()

def dict_dump(dict):
    file = open(‘core_ip_addresses.dict’, ‘w’)
    pickle.dump(dict, file)

def get_ospf_interface_ip(host, username, pwd):
    """Return the following dict:
    { ip: { host: host, interface:interface } }
    """

    get_ospf_interface_dict = {}

    dev = Device(host=host, user=username, password=pwd, normalize=True, auto_probe=4, timeout=10)

    dev.open()
    rpc = dev.rpc.get_ospf_interface_information(detail=True)
    dev.close()

    ospf_interface_list = rpc.findall(‘.//ospf-interface’)

    for interface in ospf_interface_list:
        interface_name = interface.find(‘.//interface-name’).text
        interface_address = interface.find(‘.//interface-address’).text
        get_ospf_interface_dict[interface_address] = { ‘host’: host, ‘interface’: interface_name }

    return get_ospf_interface_dict
```

```
def gather_core_ips(list_of_devices):
    """ gather the core IP addresses and return them in a dictionary"""
    core_ips_dict = {}

    for device in list_of_devices:
        print('Gathering information from device {}.'.format(device))
        try:
            addition = get_ospf_interface_ip(device, username, pwd)
            core_ips_dict.update(addition)
        except Exception as error:
            print('Ran into the following Exception when trying device {}:'.format(device))
            print(error)

    return core_ips_dict

def display_dict(dict):
    print('Here is the core_ip_dict:')
    for k ,v in dict.items():
        print('{0:20} {1:20} {2:20}'.format(v['host'], v['interface'], k))

if __name__ == "__main__":

    core_ip_dict = gather_core_ips(list_of_devices)

    dict_dump(core_ip_dict)

    display_dict(core_ip_dict)
```

This is what happens in this script:

1. It imports the standard PyEZ class `Device`. In addition to this, it also imports `pickle` to be able to store a dictionary, `os` to grab the user login name, and `getpass` to ask the user for a password.
2. When the script is run as main, the script first starts off with the following: core_ip_dict = gather_core_ips(list_of_devices). This will execute the gather_core_ips() function, taking in the list_of_devices as a parameter. In the example, the list is left empty, but this is the place where you would put all your devices. Like so, for instance: list_of_devices = [dev_1, dev_2, dev_3,]. Also, in the example we assume that we are using OSPF as an IGP. So to be able to translate the LSP EROs, we only really need to fetch the IP addresses on the interfaces that are enabled for OSPF. Finally, it will store the result in a dictionary called core_ip_dict.
3. Using dict_dump(core_ip_dict), it writes the dictionary to disk for future use.
4. Finally, it prints the content of the core_ip dictionary to screen using display_dict(core_ip_dict).

When you run the script from a Linux server, you can see the following:

```
[said@server]$ python build_lsp_dict.py
Password:
Gathering information from device ar01.mel.
Gathering information from device ar01.sjc.
<output omitted>
```

```
Here is the core_ip_dict:
<output omitted>
br02.ams   ae14.0              10.45.19.20
ar02.dal   ae12.0              10.228.118.181
ar04.fra   ae1.0               10.53.16.3
ar01.dal   ae13.0              10.228.118.185
ar02.mel   ae12.0              10.1.118.136
bbr1.hou   ae16.0              10.192.18.232
br01.hou   ae17.0              10.192.18.234
br02.mex   ae16.0              10.192.18.236
<output omitted>
```

Translating an LSP Using the Stored Information

Since we now have a dictionary that contains all the data, we can move on to the script that we will actually be using to do some troubleshooting:

```
# !/usr/bin/env python
#
# Created by Said van de Klundert
#
import getpass, os, sys, pickle
from netaddr import IPNetwork
from jnpr.junos import Device

def rpc_rsvp_session_ingress_name(username, pwd, host, lsp_name ):
    """ collects the rsvp session information and returns a list of EROs"""
    dev = Device(host=host, user=username, password=pwd, normalize=True)

    dev.open()
    rpc = dev.rpc.get_rsvp_session_information(detail = True, session_name = lsp_
name, ingress=True )
    dev.close()

    explicit_route = rpc.findall('.//explicit-route/address')

    ero_list_new = [ ip_address.text for ip_address in explicit_route ]

    return ero_list_new

def get_subnet_ips(list_of_ips):
    """
    Converts a list of IP addresses to a list that contains all the addresses for the /31 subnets
    """
    subnet_ips = []

    for ip in list_of_ips:
        link_addresses_list = []
        link_addresses_list.append(ip)
        ip_range = IPNetwork( ip + '/31')
        # finding the other IP address in the /31 subnet
        for address in ip_range:
            if str(address) != ip:
                link_addresses_list.insert(0, str(address))

        subnet_ips.extend(link_addresses_list)

    return subnet_ips

def dict_load():
    file = open('core_ip_addresses.dict', 'r')
    dict_restored = pickle.load(file)
    return dict_restored

def report_lsp(list_of_ips):
    print('{} {} traverses the following interfaces:'.format(node, lsp))
    for ip in list_of_ips:
        try:
            interface = core_ip_dict[ip]['interface']
            host = core_ip_dict[ip]['host']
            print('  {0:20} {1:20}'.format(host, interface))
        except Exception as error:
            print(error)

username = os.getlogin(); pwd = getpass.getpass()
node = str(sys.argv[1]); lsp = str(sys.argv[2])

core_ip_dict = dict_load()

if __name__ == "__main__":

    try:
        list_ero = rpc_rsvp_session_ingress_name(username, pwd, node, lsp)
    except Exception as error:
        print(error)

    try:
        list_of_ips = get_subnet_ips(list_ero)
    except Exception as error:
        print(error)
    else:
        report_lsp(list_of_ips)
```

Let's see what is happening in this script:

We import the following modules:

- `Device`: the standard PyEZ class we use to communicate with the MX router.
- `pickle` to be able to load the dictionary we stored previously.
- `os` to grab the user login name and `getpass` to ask the user for a password.
- `sys` to be able to collect two user-specified arguments for this script.
- `IPNetwork` class to be able to check the IPs involved in a /31 subnet.

When the script is run as main, the script first starts off with the following: list_ero = rpc_rsvp_session_ingress_name(username, pwd, node, lsp). This will execute the rpc_rsvp_session_ingress_name () function and store the list of IP addresses that make up the ERO in a list that is called list_ero.

This is followed by; list_of_ips = get_subnet_ips(list_ero). This function takes the list_ero list and turns that into a list that contains all addresses assuming /31 subnets are used (while maintaining the same order).

Then execute report_lsp(list_of_ips). This function will print the full path that the LSP traverses.

Finally, run the script from a Linux server. Supply the script with the required arguments (translate_lsp.py <routername> <lsp_name>) . In the following example, we are checking router ar04.fra and we are translating the LSP with the name **`10.45.17.5:dt-rsvp-AUTO-TUNNEL`** :

```
[said@server]$ python translate_lsp.py ar04.fra 10.45.17.5:dt-rsvp-AUTO-TUNNEL
Password:
ar04.fra 10.45.17.5:dt-rsvp-tunnel traverses the following interfaces:
  ar04.ams   ae1.0
  br02.fra   ae15.0
  br02.fra   ae5.0
  br01.fra   ae19.0
  br01.fra   ae2.0
  br02.osl   ae0.0
  br02.osl   ae101.0
  ar01.osl   ae1.0
```

Discussion

In this recipe, we created a dictionary that contains host/IP information, stored it on disk, grabbed the ERO for an LSP, and translated that into a list of nodes/interfaces that the LSP traverses.

Some next steps for future fun projects:

- Instead of iterating a static list in the script that gathers the IP addresses, iterate a list of devices from a file.
- Instead of just doing a lookup to see what nodes and interfaces the LSP traverses, use that very same data to log in to the router and grab the error counters of the links, or grab and analyze log messages from the router to see if any error has occurred.
- Instead of storing the data in a dictionary and saving that dictionary to disk, store the information in the DNS server and convert the translate script to do a DNS lookup to find out the hosts and interfaces involved.
- Turn the translate LSP script into something that returns data instead of screen output. Talk to the devs in your company and see if the script can become part of a more comprehensive monitoring tool.

Recipe 17: Writing eBGP Policies for Outbound Traffic Engineering

By Pierre-Yves Maunier

- Junos OS Used: 17.3R3
- Juniper Platforms General Applicability: MX Series

This recipe will help you use the power of Junos BGP policies to control the way your network traffic exits your multihomed network.

Problem

Managing a network with multiple eBGP peerings can sometimes be challenging. Adjustments are often needed to overcome running issues such as congested links, dealing with ninety-five percentile billing, routing problems, and more. If these adjustments are made with policies *not* originally designed with traffic engineering in mind, it could lead to cumbersome configurations, and possibly suboptimal routing.

NOTE This recipe only covers outbound traffic engineering (traffic exiting the network). The eBGP import policies in this recipe won't cover topics such as filtering bogons announcements or other things you should filter when you're receiving announcements from the default-free zone.

Solution

The solution is the configuration of traffic engineering-oriented eBGP policies that will help you control traffic leaving your network. First, you need to classify the various types of links you can use to exit a network. Then you'll use this information to build a clean and easy-to-read configuration that will allow you to reach our goal.

Let's have a look at Figure 17.1. There are various types of eBGP peerings to exit a network such as the IP transit providers that provide you with access to the default-free zone, peering (public, private, free, and paid), and customers.

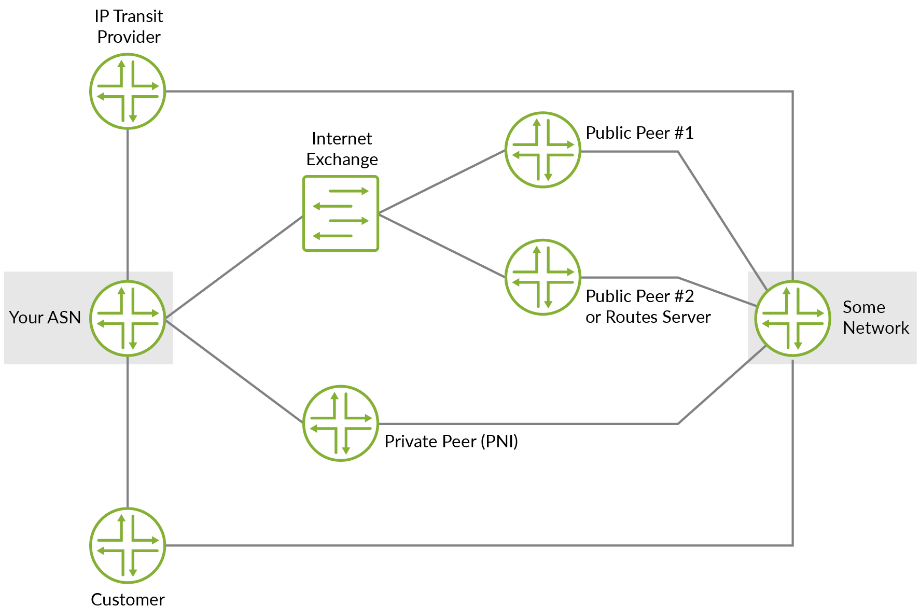


Figure 17.1 *The Main Types of eBGP Peerings*

If you receive a prefix originating from a network on all types of eBGP peerings, your router will select the best exit following certain rules listed here (a stripped-down list focusing on elements we will manipulate in this recipe's policies):

1. Highest local preference wins

2. Shortest AS path length wins

3. Lowest multiple exit discriminator (MED) wins

NOTE In order to be able to compare MEDs between different AS paths having the same length, use the `always-compare-med` statement. You can also activate the BGP multipath with the `multipath multiple-as` statement to be able to insert multiple next hops for the same route in the RIB/FIB so you can perform load-balancing between multiple equivalent exits.

Using policies to modify routes attributes to influence the route selection algorithm is called *traffic engineering*.

Looking at Figure 17.1, you want to reach "some network" prefixes using the following order if possible:

1. The customer link because the customer is usually charged for traffic using this path.

2. The private peering links as you have a direct interconnection with the peer, don't rely on an Internet Exchange platform, and they are usually free.

3. The public peering links, as they are usually cheaper than IP transit.

4. The IP transit is usually the last resort exit, as the provider charges to use it.

NOTE This order is an example based on financial considerations. Other criteria such as low latency links, quality, etc. can be taken into consideration to influence the routing. If one link is cheaper than another one but is suffering heavy packet loss, depending on the criticality of the traffic, you may want to use a more expensive and higher quality link.

An easy way to achieve the traffic engineering rules above would be to just set a local preference value on each eBGP session to indicate a preference. For example in Table 17.1 are some local preferences.

Table 17.1 *Basic Traffic Engineering Local Preferences Setup*

Peer Type	Local Preference
IP Transit Provider	1000
Peer	2000
Customer	3000

This is a bit simplistic, and you want to have more versatile routing policies that will allow you to:

- Have a distinction between a private peer or a public peer,
- Filter (reject) a prefix, an ASN, or an AS Path from a peer announcement (so the routes are not installed in the routing table),
- Force a prefix, an ASN, or an AS Path to an IP transit provider even if the route was also received on a peer or customer link.

There are two methods to influence how the traffic is exiting your network, either:

- You force the traffic to go to a specific eBGP peer, or
- You prevent the traffic from going to a specific eBGP peer.

Both ways are good, but depending on the situation you may prefer one or the other. In any case, it's best if your policies can handle both.

Okay, let's update Table 17.1 with more traffic engineering goals that will help us to configure our routing policies.

NOTE To maintain clarity and simplify the configuration in this recipe we will only consider IP transit providers and peers. We will leave the customer traffic engineering rules as an exercise for the reader.

Table 17.2 Advanced Traffic Engineering Local Preferences Setup

Peer Type	Goal	Local Preference
Peer	Depref prefix or AS Path below IP Transit	700
IP Transit	Depref prefix below other IP Transits	800
IP Transit	Depref AS Path below other IP Transits	900
IP Transit	Normal IP Transit local preference	1000
IP Transit	Prefer AS Path over other IP Transits	1100
IP Transit	Prefer prefix over other IP Transits	1200
Peer	Depref prefix below other Peers	1800
Peer	Depref AS Path below other Peers	1900
Peer	Normal Peer local preference	2000
Peer	Prefer AS Path over other Peers	2100
Peer	Prefer prefix over other Peers	2200
IP Transit	Prefer AS Path over Peers	2300
IP Transit	Prefer prefix over Peers	2400

If you look at Table 17.2, you can see that there is no distinction between private peers, public peers, and Internet Exchange route servers. Let's change that using the MEDs.

Table 17.3 Advanced Traffic Engineering MED Setup

Peer Type	MED
IP Transit	100
Private Peering	90
Public Peering	100
Public Peering route server	110

Table 17.3 allows us to send traffic primarily to a private peer if we receive the same prefix *with the same AS path length* from multiple peers. See the topology lab setup in Figure 17.1.

NOTE *With the same AS path length* is important here. As the AS path length is evaluated before the MED in the BGP route selection algorithm, this setup allows you to prefer private peers unless there is a prefix with a shorter AS path length coming from another peer. This is the author of this recipe's personal preference, as generally, the less ASNs your traffic goes through, the better. This also depends on the relationship you may have with the private peer. Some networks use a higher local preference on private peers compared to public peers, and in that case, even if there is a shorter route on a public peer, the traffic will be sent to the private peer on the longer AS path route.

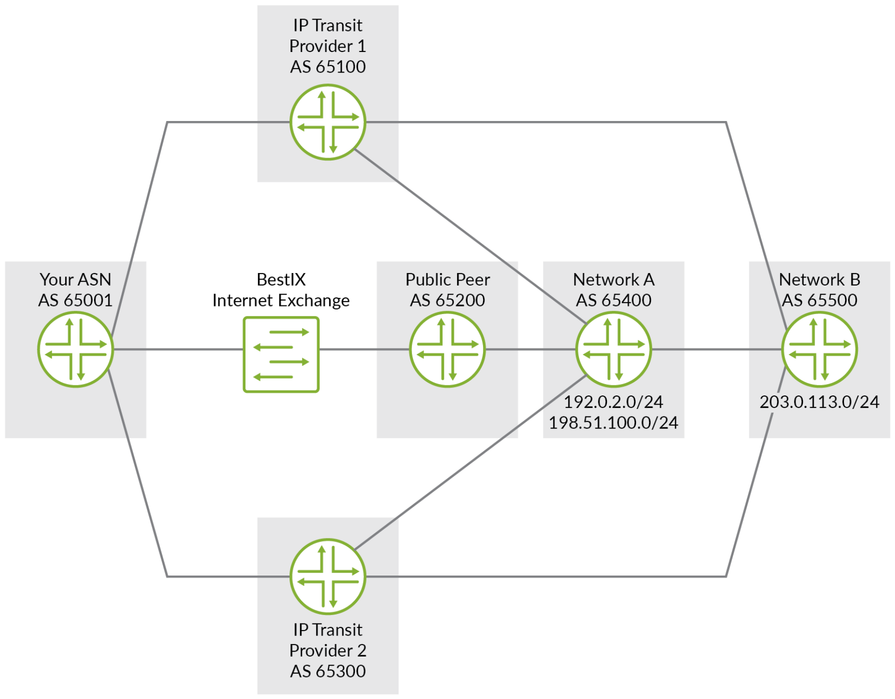


Figure 17.2 Lab Setup for Recipe 17

Let's do a very basic BGP configuration on AS65001's router:

```
set protocols bgp path-selection always-compare-med
set protocols bgp group v4-transit neighbor 10.0.0.1 description "Transit 1"
set protocols bgp group v4-transit neighbor 10.0.0.1 peer-as 65100
set protocols bgp group v4-transit neighbor 10.0.0.5 description "Transit 2"
set protocols bgp group v4-transit neighbor 10.0.0.5 peer-as 65300
set protocols bgp group v4-peer-bestix neighbor 10.0.0.3 description "Public Peer"
set protocols bgp group v4-peer-bestix neighbor 10.0.0.3 peer-as 65200
```

This configures two BGP groups, one for all the IP Transit providers, and one for all peers on the BestIX Internet exchange.

Now let's see how to reach Network B prefix 203.0.113.0/24:

```
lab@as65000> show route 203.0.113.0/24

inet.0: 12 destinations, 19 routes (12 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

203.0.113.0/24     *[BGP/170] 1d 17:04:18, localpref 100
                      AS path: 65100 65500 I, validation-state: unverified
                    > to 10.0.0.1 via xe-0/0/1.0
                    [BGP/170] 1d 17:04:18, localpref 100
                      AS path: 65300 65500 I, validation-state: unverified
                    > to 10.0.0.5 via xe-0/3/1.0
                    [BGP/170] 1d 17:04:18, localpref 100
                      AS path: 65200 65400 65500 I, validation-state: unverified
                    > to 10.0.0.3 via xe-0/0/2.0
```

Without any import policy applied on the BGP peers, a default local preference of 100 is applied to all peers and the best path is via IP transit 1 because the AS path length is smaller than on the public peer on the Internet exchange. Why is IP Transit 1 preferred over IP Transit 2 ? The route is exactly the same and has the same parameters and even the age of the route is the same, so the breakdown comes to IP Transit 1 having a lower peer IP address than IP Transit 2. Let's create import policies that will allow you to force traffic via the public peer and also add lots of fun ways to move the traffic around.

NOTE If we configure the `multipath multiple-as` statement in the IPv4-transit BGP group, the traffic will be load-balanced between IP Transit 1 and IP Transit 2 (two next hops will be installed in the FIB for this route).

Let's configure the IP transit 1 import policy:

```
set policy-options policy-statement v4-transit1-in term 100 from as-path-group as-filter-on-transit1
set policy-options policy-statement v4-transit1-in term 100 then reject
set policy-options policy-statement v4-transit1-in term 110 from policy v4-pfx-filter-on-transit1
set policy-options policy-statement v4-transit1-in term 110 then reject
set policy-options policy-statement v4-transit1-in term 200 from policy ( slash8-24 && v4-pfx-
depref-to-transit1 )
set policy-options policy-statement v4-transit1-in term 200 then metric 100
set policy-options policy-statement v4-transit1-in term 200 then local-preference 800
set policy-options policy-statement v4-transit1-in term 200 then accept
set policy-options policy-statement v4-transit1-in term 210 from as-path-group as-depref-to-transit1
set policy-options policy-statement v4-transit1-in term 210 from policy slash8-24
set policy-options policy-statement v4-transit1-in term 210 then metric 100
set policy-options policy-statement v4-transit1-in term 210 then local-preference 900
set policy-options policy-statement v4-transit1-in term 210 then accept
set policy-options policy-statement v4-transit1-in term 220 from policy ( slash8-24 && v4-pfx-pref-
to-transit1-over-peer )
set policy-options policy-statement v4-transit1-in term 220 then metric 100
set policy-options policy-statement v4-transit1-in term 220 then local-preference 2400
set policy-options policy-statement v4-transit1-in term 220 then accept
set policy-options policy-statement v4-transit1-in term 230 from as-path-group as-pref-to-transit1-
over-peer
set policy-options policy-statement v4-transit1-in term 230 from policy slash8-24
set policy-options policy-statement v4-transit1-in term 230 then metric 100
set policy-options policy-statement v4-transit1-in term 230 then local-preference 2300
set policy-options policy-statement v4-transit1-in term 230 then accept
set policy-options policy-statement v4-transit1-in term 240 from policy ( slash8-24 && v4-pfx-pref-
to-transit1 )
set policy-options policy-statement v4-transit1-in term 240 then metric 100
set policy-options policy-statement v4-transit1-in term 240 then local-preference 1200
set policy-options policy-statement v4-transit1-in term 240 then accept
set policy-options policy-statement v4-transit1-in term 250 from as-path-group as-pref-to-transit1
set policy-options policy-statement v4-transit1-in term 250 from policy slash8-24
set policy-options policy-statement v4-transit1-in term 250 then metric 100
set policy-options policy-statement v4-transit1-in term 250 then local-preference 1100
set policy-options policy-statement v4-transit1-in term 250 then accept
set policy-options policy-statement v4-transit1-in term 300 from policy slash8-24
set policy-options policy-statement v4-transit1-in term 300 then metric 100
set policy-options policy-statement v4-transit1-in term 300 then local-preference 1000
set policy-options policy-statement v4-transit1-in term 300 then accept
set policy-options policy-statement v4-transit1-in then reject
```

Let's break it down and look at each term individually to see what they do and how they do it.

Term 100 : This term will allow us to not accept a route with a specific AS path. We use an `as-path-group` in order to be able to use multiple AS paths in the same term. Routes matching term 100 will not be installed in the routing table.

Here is the configuration of this AS path-group :

```
set policy-options as-path-group as-filter-on-transit1 as-path dummy 65000
```

You can't have an empty AS path-group, so configure it with a dummy AS path (you should never receive a private ASN on a public eBGP session).

Term 110 : This is similar to `term 100` but instead of matching an AS path we're matching prefixes:

```
set policy-options policy-statement v4-pfx-filter-on-transit1 term 1 from route-
filter 0.0.0.0/32 exact
set policy-options policy-statement v4-pfx-filter-on-transit1 term 1 then accept
set policy-options policy-statement v4-pfx-filter-on-transit1 then reject
```

There's also a dummy prefix in this policy.

Term 200 : This is the first term where we actually accept prefixes in the routing table. If a prefix is matched in the `v4-pfx-depref-to-transit1` policy, its local preference will be set to 800 (see Table 17.2) to make sure this route will not be preferred over another IP transit provider. The `slash8-24` is just a policy to make sure we only accept routes that are between /8 and /24 (in IPv4, other prefix sizes should not be present in the default-free zone, so we're dropping them if we see them):

```
set policy-options policy-statement v4-pfx-depref-to-transit1 term 1 from route-
filter 0.0.0.0/32 exact
set policy-options policy-statement v4-pfx-depref-to-transit1 term 1 then accept
set policy-options policy-statement v4-pfx-depref-to-transit1 then reject

set policy-options policy-statement slash8-24 term 1 from route-filter 0.0.0.0/0 prefix-length-
range /8-/24
```

```
set policy-options policy-statement slash8-24 term 1 then accept
set policy-options policy-statement slash8-24 then reject
```

NOTE Every `prefix-list` named v4-pfx-pref-xxx, v4-pfx-depref-xxx, and v4-pfx-filter-xxx have the same format as described above. Every `as-path-group` named as-pref-xxx, as-depref-xxx, and as-filter-xxx has the same format as described above. We won't detail the next ones.

You get the idea, the other terms are pretty self-explanatory and you can use Table 17.2 to get the goal of each term (by comparing the local-preferences).

Duplicate this policy for every other IP transit provider and don't forget to create specific prefix lists and AS path-groups for each IP transit provider.

Let's have a look at the import policy for public peers on the BestIX Internet exchange:

```
set policy-options policy-statement v4-peer-bestix-in term 100 from as-path-group as-filter-on-
bestix
set policy-options policy-statement v4-peer-bestix-in term 100 then reject
set policy-options policy-statement v4-peer-bestix-in term 110 from policy v4-pfx-filter-on-bestix
set policy-options policy-statement v4-peer-bestix-in term 110 then reject
set policy-options policy-statement v4-peer-bestix-in term 180 from policy ( slash8-24 && v4-pfx-
depref-to-bestix-below-transit )
set policy-options policy-statement v4-peer-bestix-in term 180 then metric 100
set policy-options policy-statement v4-peer-bestix-in term 180 then local-preference 700
set policy-options policy-statement v4-peer-bestix-in term 180 then accept
set policy-options policy-statement v4-peer-bestix-in term 190 from as-path-group as-depref-to-
bestix-below-transit
set policy-options policy-statement v4-peer-bestix-in term 190 from policy slash8-24
set policy-options policy-statement v4-peer-bestix-in term 190 then metric 100
set policy-options policy-statement v4-peer-bestix-in term 190 then local-preference 700
set policy-options policy-statement v4-peer-bestix-in term 190 then accept
set policy-options policy-statement v4-peer-bestix-in term 200 from policy ( slash8-24 && v4-pfx-
depref-to-bestix )
set policy-options policy-statement v4-peer-bestix-in term 200 then metric 100
set policy-options policy-statement v4-peer-bestix-in term 200 then local-preference 1800
set policy-options policy-statement v4-peer-bestix-in term 200 then accept
set policy-options policy-statement v4-peer-bestix-in term 210 from as-path-group as-depref-to-
bestix
set policy-options policy-statement v4-peer-bestix-in term 210 from policy slash8-24
set policy-options policy-statement v4-peer-bestix-in term 210 then metric 100
set policy-options policy-statement v4-peer-bestix-in term 210 then local-preference 1900
set policy-options policy-statement v4-peer-bestix-in term 210 then accept
set policy-options policy-statement v4-peer-bestix-in term 240 from policy ( slash8-24 && v4-pfx-
pref-to-bestix )
set policy-options policy-statement v4-peer-bestix-in term 240 then metric 100
set policy-options policy-statement v4-peer-bestix-in term 240 then local-preference 2200
set policy-options policy-statement v4-peer-bestix-in term 240 then accept
set policy-options policy-statement v4-peer-bestix-in term 250 from as-path-group as-pref-to-bestix
set policy-options policy-statement v4-peer-bestix-in term 250 from policy slash8-24
set policy-options policy-statement v4-peer-bestix-in term 250 then metric 100
set policy-options policy-statement v4-peer-bestix-in term 250 then local-preference 2100
set policy-options policy-statement v4-peer-bestix-in term 250 then accept
set policy-options policy-statement v4-peer-bestix-in term 300 from policy slash8-24
set policy-options policy-statement v4-peer-bestix-in term 300 then metric 100
set policy-options policy-statement v4-peer-bestix-in term 300 then local-preference 2000
set policy-options policy-statement v4-peer-bestix-in term 300 then accept
set policy-options policy-statement v4-peer-bestix-in then reject
```

This policy is similar to those in use for the IP transit, the main differences being:

- The local preferences are higher on the public peerings to indicate a preference for routes learned on these BGP sessions.
- There is a term on the IP transit policy to set a higher local preference than public peers when there is a need to force traffic to exit through the transits.
- There is a term on the public peering policy to set a lower local preference than IP transits when there is a need to avoid sending traffic over a public peer.

NOTE For a private peering, the policy will be exactly the same but with a MED of 90 applied on all routes, and a route server peering the policy will have a MED of 110, see Table 17.3.

This is how we reached 203.0.113.0/24 before applying the policies:

```
lab@as65000> show route 203.0.113.0/24

inet.0: 12 destinations, 19 routes (12 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

203.0.113.0/24     *[BGP/170] 1d 17:04:18, localpref 100
                      AS path: 65100 65500 I, validation-state: unverified
                    > to 10.0.0.1 via xe-0/0/1.0
                    [BGP/170] 1d 17:04:18, localpref 100
                      AS path: 65300 65500 I, validation-state: unverified
                    > to 10.0.0.5 via xe-0/3/1.0
                    [BGP/170] 1d 17:04:18, localpref 100
                      AS path: 65200 65400 65500 I, validation-state: unverified
                    > to 10.0.0.3 via xe-0/0/2.0
```

Now let’s apply the policies to the peers:

```
set protocols bgp group v4-transit neighbor 10.0.0.1 import v4-transit1-in
set protocols bgp group v4-transit neighbor 10.0.0.5 import v4-transit2-in
set protocols bgp group v4-peer-bestix import v4-peer-bestix-in
```

And after committing the configuration:

```
lab@as65000> show route 203.0.113.0/24

inet.0: 12 destinations, 19 routes (12 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

203.0.113.0/24     *[BGP/170] 00:00:12, MED 100, localpref 2000
                      AS path: 65200 65400 65500 I, validation-state: unverified
                    > to 10.0.0.3 via xe-0/0/2.0
                    [BGP/170] 00:00:12, MED 100, localpref 1000
                      AS path: 65100 65500 I, validation-state: unverified
                    > to 10.0.0.1 via xe-0/0/1.0
                    [BGP/170] 00:00:12, MED 100, localpref 1000
                      AS path: 65300 65500 I, validation-state: unverified
                    > to 10.0.0.5 via xe-0/3/1.0
```

This is a very basic approach to traffic engineering: a preference of peers over transits. But sometimes you may need more granularity.

Let's say that you have issues reaching Network B (AS65500) because you suspect saturation and packet loss on the public peer's (AS65200) network interface facing the Internet exchange, and you want to avoid this route.

You can try this:

```
set policy-options policy-statement v4-pfx-depref-to-bestix-below-transit term 1 from route-
filter 203.0.113.0/24 exact
```

And this is how that affects the routing table:

```
lab@as65000> show route 203.0.113.0/24

inet.0: 12 destinations, 19 routes (12 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

203.0.113.0/24     *[BGP/170] 00:05:57, MED 100, localpref 1000
                      AS path: 65100 65500 I, validation-state: unverified
                    > to 10.0.0.1 via xe-0/0/1.0
                    [BGP/170] 00:05:57, MED 100, localpref 1000
                      AS path: 65300 65500 I, validation-state: unverified
                    > to 10.0.0.5 via xe-0/3/1.0
                    [BGP/170] 00:00:42, MED 100, localpref 700
                      AS path: 65200 65400 65500 I, validation-state: unverified
                    > to 10.0.0.3 via xe-0/0/2.0
```

You can see that the route to reach 203.0.113.0/24 is now active (shown with a *) over Transit 1 because the local preference of the route via the public peer has been lowered from 2000 to 700.

You can confirm by looking at the route received via the peering with AS65200 using `detail`:

```
lab@as65000> show route 203.0.113.0/24 next-hop 10.0.0.3 detail | find inactive
                Inactive reason: Local Preference
                Local AS: 65001 Peer AS: 65200
                Age: 5:11   Metric: 100
                Validation State: unverified
                Task: BGP_65200.10.0.0.3+179
                AS path: 65200 65400 65500 I
                Accepted
                Localpref: 700
                Router ID: 10.0.0.3
```

So we've solved the issue for 203.0.113.0/24, but what about every other network that we currently reach via AS65200 that's having an issue on the Internet exchange?

Let's see the whole routing table:

```
lab@as65000> show route protocol bgp

inet.0: 12 destinations, 19 routes (12 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

192.0.2.0/24       *[BGP/170] 01:45:58, MED 100, localpref 2000
                      AS path: 65200 65400 I, validation-state: unverified
                    > to 10.0.0.3 via xe-0/0/2.0
                    [BGP/170] 01:45:58, MED 100, localpref 1000
                      AS path: 65100 65400 I, validation-state: unverified
                    > to 10.0.0.1 via xe-0/0/1.0
                    [BGP/170] 01:45:58, MED 100, localpref 1000
                      AS path: 65300 65400 I, validation-state: unverified
                    > to 10.0.0.5 via xe-0/3/1.0
198.51.100.0/24    *[BGP/170] 01:45:58, MED 100, localpref 2000
                      AS path: 65200 65400 I, validation-state: unverified
                    > to 10.0.0.3 via xe-0/0/2.0
                    [BGP/170] 01:45:58, MED 100, localpref 1000
                      AS path: 65100 65400 I, validation-state: unverified
                    > to 10.0.0.1 via xe-0/0/1.0
                    [BGP/170] 01:45:58, MED 100, localpref 1000
                      AS path: 65300 65400 I, validation-state: unverified
                    > to 10.0.0.5 via xe-0/3/1.0
203.0.113.0/24     *[BGP/170] 01:45:58, MED 100, localpref 1000
                      AS path: 65100 65500 I, validation-state: unverified
                    > to 10.0.0.1 via xe-0/0/1.0
                    [BGP/170] 01:45:58, MED 100, localpref 1000
                      AS path: 65300 65500 I, validation-state: unverified
                    > to 10.0.0.5 via xe-0/3/1.0
                    [BGP/170] 01:40:43, MED 100, localpref 700
                      AS path: 65200 65400 65500 I, validation-state: unverified
                    > to 10.0.0.3 via xe-0/0/2.0
```

You can see that 192.0.2.0/24 and 198.51.100.0/24 are still preferred over the peering session with AS65200. You could do the same as before: add these prefixes in the `v4-pfx-depref-to-bestix-below-transit` policy, but this won't take into account any routes you may receive from AS65200 in the future. This is where the `as-path-group` can help.

Let's remove our previous configuration:

```
delete policy-options policy-statement v4-pfx-depref-to-bestix-below-transit term 1 from route-
filter 203.0.113.0/24 exact
```

Update the `as-path-group` this way:

```
set policy-options as-path-group as-depref-to-bestix-below-transit as-path "Public Peer" 65200.*
```

And see how it affected the routing table:

```
lab@as65000> show route protocol bgp

inet.0: 12 destinations, 19 routes (12 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both
```

```
192.0.2.0/24       *[BGP/170] 02:40:02, MED 100, localpref 1000
                      AS path: 65100 65400 I, validation-state: unverified
                    > to 10.0.0.1 via xe-0/0/1.0
                    [BGP/170] 02:40:02, MED 100, localpref 1000
                      AS path: 65300 65400 I, validation-state: unverified
                    > to 10.0.0.5 via xe-0/3/1.0
                    [BGP/170] 00:18:44, MED 100, localpref 700
                      AS path: 65200 65400 I, validation-state: unverified
                    > to 10.0.0.3 via xe-0/0/2.0
198.51.100.0/24    *[BGP/170] 02:40:02, MED 100, localpref 1000
                      AS path: 65100 65400 I, validation-state: unverified
                    > to 10.0.0.1 via xe-0/0/1.0
                    [BGP/170] 02:40:02, MED 100, localpref 1000
                      AS path: 65300 65400 I, validation-state: unverified
                    > to 10.0.0.5 via xe-0/3/1.0
                    [BGP/170] 00:18:44, MED 100, localpref 700
                      AS path: 65200 65400 I, validation-state: unverified
                    > to 10.0.0.3 via xe-0/0/2.0
203.0.113.0/24     *[BGP/170] 02:40:02, MED 100, localpref 1000
                      AS path: 65100 65500 I, validation-state: unverified
                    > to 10.0.0.1 via xe-0/0/1.0
                    [BGP/170] 02:40:02, MED 100, localpref 1000
                      AS path: 65300 65500 I, validation-state: unverified
                    > to 10.0.0.5 via xe-0/3/1.0
                    [BGP/170] 00:18:44, MED 100, localpref 700
                      AS path: 65200 65400 65500 I, validation-state: unverified
                    > to 10.0.0.3 via xe-0/0/2.0
```

Even if the same policy is applied to every peer on the Internet exchange, that AS Path group will only match routes received from AS65200, so this is pretty handy in this situation.

Let's take the exercise a bit further: from the routing table you can see that all the traffic is flowing through IP Transit 1 (AS65100). Everything now looks okay from AS65001's perspective to reach Network B (AS65500), but you have some issues reaching Network A (AS65400).

To troubleshoot this, you want to force the traffic to one of the Network A prefixes via IP transit 2 to compare the performance between IP Transit 1 and IP Transit 2.

Let's force 192.0.2.0/24 to IP Transit 2:

```
set policy-options policy-statement v4-pfx-pref-to-transit2 term 1 from route-
filter 192.0.2.0/24 exact
```

And see how this affects the routing table:

```
lab@as65000# run show route protocol bgp

inet.0: 12 destinations, 19 routes (12 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

192.0.2.0/24       *[BGP/170] 00:00:23, MED 100, localpref 1200
                      AS path: 65300 65400 I, validation-state: unverified
                    > to 10.0.0.5 via xe-0/3/1.0
                    [BGP/170] 02:49:35, MED 100, localpref 1000
```

```
                      AS path: 65100 65400 I, validation-state: unverified
                    > to 10.0.0.1 via xe-0/0/1.0
                    [BGP/170] 00:28:17, MED 100, localpref 700
                      AS path: 65200 65400 I, validation-state: unverified
                    > to 10.0.0.3 via xe-0/0/2.0
198.51.100.0/24    *[BGP/170] 02:49:35, MED 100, localpref 1000
                      AS path: 65100 65400 I, validation-state: unverified
                    > to 10.0.0.1 via xe-0/0/1.0
                    [BGP/170] 02:49:35, MED 100, localpref 1000
                      AS path: 65300 65400 I, validation-state: unverified
                    > to 10.0.0.5 via xe-0/3/1.0
                    [BGP/170] 00:28:17, MED 100, localpref 700
                      AS path: 65200 65400 I, validation-state: unverified
                    > to 10.0.0.3 via xe-0/0/2.0
203.0.113.0/24     *[BGP/170] 02:49:35, MED 100, localpref 1000
                      AS path: 65100 65500 I, validation-state: unverified
                    > to 10.0.0.1 via xe-0/0/1.0
                    [BGP/170] 02:49:35, MED 100, localpref 1000
                      AS path: 65300 65500 I, validation-state: unverified
                    > to 10.0.0.5 via xe-0/3/1.0
                    [BGP/170] 00:28:17, MED 100, localpref 700
                      AS path: 65200 65400 65500 I, validation-state: unverified
                    > to 10.0.0.3 via xe-0/0/2.0
```

Now the local preference for 192.0.2.0/24 is 1200 on the route learned from IP Transit 2.

There's a lot of ways to play around with these policies, and even with these policies it can sometimes be difficult to track what is configured in which prefix-list policy or AS path group. For example, a friend of mine wrote an Op-script that parses the configuration and gives you a bird's eye view of your traffic engineering configuration:

```
lab@as65000> op show-routing-eng

Checking prefix list :
    Prefix               in Policy
----------------------------------------
192.0.2.0/24         v4-pfx-pref-to-transit2


Checking as-path-group :
    Name                 AS path                 in As-path-group
------------------------------------------------------------------------
Public Peer          65200.*              as-depref-to-bestix-below-transit
```

Here's the script source code if you want to play with it:

```
lab@as65000> file show /var/db/scripts/op/show-routing-eng.slax
version 1.2;
ns junos = «http://xml.juniper.net/junos/*/junos»;
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns str = "http://exslt.org/strings";
import "../import/junos.xsl";
```

```
match / {
    <op-script-results> {
    /* API Element to retrieve the policy-options configuration in XML */
        var $config-rpc = {
            <get-configuration database="committed"> {
                <configuration> {
                    <policy-options>;
                    }
                }
            }

        var $result = jcs:invoke( $config-rpc );
        var $pfxpattern = "v4-pfx";
        var $ippattern = "0\\.0\\.0\\.0";
        var $pfxformat = "%-20s %-20s";
        expr jcs:output("\nChecking prefix list : \r");
        expr jcs:output(jcs:printf($pfxformat, "    Prefix", "    in Policy"));
        expr jcs:output( str:padding(40, "-"));
        for-each ($result/policy-options/policy-statement/name[starts-with(., $pfxpattern)]) {
            var $context = ../name;
            for-each (../descendant::route-filter/address) {
                if ( not ( jcs:regex ($ippattern, .))) {
                    expr jcs:output(jcs:printf($pfxformat, ., $context));
                }
            }
        }
        expr jcs:output("\n\nChecking as-path-group : \r");
        var $asgpattern1 = "as-pref";
        var $asgpattern2 = "as-depref";
        var $asgpattern3 = "as-filter";
        var $aspattern = "65000";
        var $asformat = "%-20s %-20s %-32s";
        expr jcs:output(jcs:printf($asformat, "    Name", "   AS path", "      in As-path-group"));
        expr jcs:output(str:padding(72, "-"));
        for-each ($result/policy-options/as-path-group/name[starts-with(., $asgpattern1) or starts-
with(., $asgpattern2) or starts-with(., $asgpattern3)]) {
            var $context = ../name;
            for-each (../descendant::path) {
                if ( not ( jcs:regex ($aspattern, .))) {
                    expr jcs:output(jcs:printf($asformat, ../name,  ., $context));
                }
            }
        }
    }
}
```

This recipe should help you ease your traffic engineering processes and help you troubleshoot and solve problems you may encounter.

Discussion

This recipe was all about you controlling the way the traffic exits your network and we played with local preference, metric, and accepting (or not) routes in the routing table. We could also have played with AS PATH prepend and you may need this flexibility in your configurations. It's up to you to play with it and modify these policies.

We did not talk about controlling the inbound traffic in this recipe, but by using the same type of policies (export this time and not import) you can achieve traffic engineering and change the way you announce your routes to your peers by modifying the AS PATH length, de-aggregating some prefixes (not a best practice as it adds entries into the global routing table but sometimes it's necessary), sending MEDs to your peers if they listen to them, etc. You can also play with BGP communities to let your customers influence the way their prefixes behave in your network, thus doing a bit of traffic engineering as well.

Recipe 18: Synchronizing Junos Device Configurations Using Python Scripts

By Peter Klimai

- Python Version Used: 2.7
- Junos PyEZ Version Used: 2.1.0
- Junos OS Used: 18.3
- Juniper Platforms General Applicability: All

This recipe shows you how you can use Python to synchronize Junos configurations between multiple devices. Interestingly, you can run the same script either directly from the Junos box, or from a Linux management server, with the same result!

Problem

In many cases, the same configuration must be applied to multiple Junos devices in your network. There are different ways to approach this task, and this recipe shows you one of them.

Solution

You mark certain parts of the configuration with a special flag on one of your devices. Then, you develop and use the Python script that copies those parts of the configuration to a set of other devices.

Let's say you have a prefix list named ALLOWED_IPS and you want to synchronize (copy) it to multiple devices:

```
[edit]
lab@vMX-1# show policy-options
prefix-list ALLOWED_IPS {
    10.254.0.1/32;
    10.254.0.2/32;
    10.254.0.3/32;
}
```

The script that you use is not limited to synchronizing only prefix lists. In fact, it can synchronize any configuration parts, but we'll use prefix lists here just as an example.

Let's mark the above part of configuration with a special "flag" first. Junos has a capability of adding a hidden `apply-macro` statement at any level of configuration hierarchy, so let's use it, and let the macro name be *SYNC*:

```
[edit]
lab@vMX-1# set policy-options prefix-list ALLOWED_IPS apply-macro SYNC

[edit]
lab@vMX-1# show | compare
[edit policy-options prefix-list ALLOWED_IPS]
+    apply-macro SYNC;

[edit]
lab@vMX-1# commit
commit complete
```

NOTE When you type in the apply-macro statement, the Junos CLI auto-complete feature will not work: you need to type the statement in full.

NOTE The original intent for introducing apply-`macro` statements in Junos was to allow users to create their custom configuration syntax in Junos with the help of commit scripts, but other uses are not prohibited. Consult the following URL for more details on custom configuration syntax: https://www.juniper.net/documentation/en_US/junos/topics/concept/junos-software-automation-commit-script-macros.html.

Let's also view the above configuration in the XML form – that will help:

```
[edit]
lab@vMX-1# show policy-options prefix-list ALLOWED_IPS | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/18.3R1/junos">
    <configuration junos:changed-seconds="1518422336" junos:changed-
localtime="2018-02-12 07:58:56 UTC">
            <policy-options>
                <prefix-list>
                    <name>ALLOWED_IPS</name>
                    <apply-macro>
                        <name>SYNC</name>
                    </apply-macro>
                    <prefix-list-item>
                        <name>10.254.0.1/32</name>
                    </prefix-list-item>
                    <prefix-list-item>
                        <name>10.254.0.2/32</name>
                    </prefix-list-item>
                    <prefix-list-item>
                        <name>10.254.0.3/32</name>
                    </prefix-list-item>
                </prefix-list>
            </policy-options>
    </configuration>
    <cli>
        <banner>[edit]</banner>
    </cli>
</rpc-reply>
```

From this output, you can see that the `apply-macro` statement transforms into the `<apply-macro>` XML element with the `<name>` child node indicating the macro's name.

The task for the Python script, at a very high level, is as follows:

1. Determine the host to retrieve the configuration (HOST_FROM). In case the script is executed on a Junos device, use the local device as HOST_FROM, otherwise ask the user for its address and login credentials.

2. On the HOST_FROM, look for all entries of apply-macro SYNC statement in the configuration, and save all such parts of the configuration.

3. Load the saved part(s) of configuration to target hosts (HOSTS_TO). Use the load replace method, so that existing objects with the same name on the target hosts are overridden.

One of the goals here is to develop a rather universal script that can work on-box and off-box without modification. Because currently (Junos 18.x timeframe) only Python 2.7 language is supported on-box, we'll use Python 2.7 for the programming language. The script will not work with Python 3 because of some small differences between language versions (e.g., `raw_input` function and `print` statement are used in a way that is specific to Python 2).

Next the complete source code of the `jsync.py` script that is solving the above task is shown, and a detailed explanation of the code follows. The script also contains some comments inside (lines starting with **#** character) that should help you figure out how it works:

```
#!/usr/bin/python2.7

# ==(1)== Perform imports =====

# Import Junos PyEZ Device and Config classes, and exceptions
from jnpr.junos import Device
from jnpr.junos.utils.config import Config
from jnpr.junos.exception import *

# Import required lxml objects to work with XML data
from lxml.etree import ElementTree, tostring


# ==(2)== Define some values for use later =====

# The name of apply-macro that is used to tag parts of configuration
# you want to synchronize to other devices
```

```
SYNC_MACRO = "SYNC"

# Script version
VERSION = "0.1"

# Set of remote (destination) hosts to synchronize the configuration
HOSTS_TO_SYNC = [ "10.254.0.42", "10.254.0.35", "10.254.0.37" ]

# Timeout for auto-probe feature
HOSTS_TO_TIMEOUT = 10


# ==(3)== Define the main function =====

def main():

    print "*** jsync script version {} started ***".format(VERSION)


    # ==(4)== Test if the script works on-box or off-box =====

    # Here and below, try/except blocks are used to process different
    # possible exceptional situations
    try:
        # If the below imports work, we are on-box
        from junos import Junos_Context
        from jcs import get_secret as get_password_securely
        print "*** Working in on-box mode ***"
        host_from = None
        user_from = None
        passwd_from = None
    except ImportError:
        # Else, working off-box
        print "*** Working in off-box mode ***"
        host_from = raw_input("Enter host to retrieve configuration (HOST_FROM): ")
        user_from = raw_input("Username on the HOST_FROM: ")

        # Also, in this case we will use getpass for password input
        from getpass import getpass as get_password_securely
        passwd_from = get_password_securely("Password on the HOST_FROM: ")


    # ==(5)== Read configuration from HOST_FROM device =====

    try:
        with Device(host=host_from, user=user_from, passwd=passwd_from) as dev:
            # Read configuration using Junos PyEZ RPC call
            conf = dev.rpc.get_configuration();
    except ConnectError as error:
        print "Error: can't connect to host {} due to {}".format(host_from, str(error))
        print "Exiting!"
        return
    except RpcError as error:
        print "Error: can't read configuration from host {} due to {}" \
                .format(host_from, str(error))
        print "Exiting!"
        return
```

```
# ==(6)== Extract configuration items to sync =====

# Build an element tree object - needed for getpath functionality
tree = ElementTree(conf)

# Start building XML delta configuration
conf_to_sync = "<configuration>\n"

# Search for all configuration stanzas that have apply-macro `SYNC_MACRO`
# and iterate over that list
for el in tree.xpath("//*[apply-macro[name='{}']]".format(SYNC_MACRO)):
    # Add replace="replace" attribute
    el.attrib["replace"]="replace"

    # Get configuration stanza
    # Example str_path result: '/configuration/policy-options/prefix-list'
    str_path = tree.getpath(el)

    # Now we will split str_path using '/' as delimiter to form a list.
    # Example: after splitting '/configuration/policy-options/prefix-list'
    # we get ['', 'configuration', 'policy-options', 'prefix-list'],
    # and we only need elements from [2] until the end of the list, not
    # including the last element (which is [-1]). So in this example,
    # list_path will be assigned value of ['policy-options'].
    list_path = str_path.split("/")[2:-1]

    # Add opening XML tags for all levels of hierarchy
    for path_el in list_path:
        conf_to_sync += "<{}>\n".format(path_el)

    # Add XML configuration part we want to synchronize
    conf_to_sync += tostring(el).strip() + '\n'

    # Add closing XML tags for all levels of hierarchy (reversed order)
    for path_el in reversed(list_path):
        conf_to_sync += "</{}>\n".format(path_el)

# Finalize XML doc - final closing tag
conf_to_sync += "</configuration>\n"


# ==(7)== Display config to the user and ask for credentials on HOSTS_TO_SYNC =====

print "The config to sync is:"
print conf_to_sync
print
print "Will sync this config to a following set of hosts:"
for host in HOSTS_TO_SYNC:
    print "  -   {}".format(host)

user_on_to_hosts = raw_input("Username on the HOSTS_TO_SYNC: ")
passwd_on_to_hosts = get_password_securely("Password on the HOSTS_TO_SYNC: ")


# ==(8)== Load configs on HOSTS_TO_SYNC =====

for host in HOSTS_TO_SYNC:
        print
        print "Working on host {}".format(host)
        try:
            # Create a PyEZ Device instance and open a NETCONF connection
            with Device(host=host, user=user_on_to_hosts, passwd=passwd_on_to_hosts,
                    auto_probe=HOSTS_TO_TIMEOUT) as dev:
                try:
                    # Create a PyEZ Config object and lock configuration
                    with Config(dev, mode="exclusive") as conf:
                        # Load configuration on the host
                        conf.load(conf_to_sync)
                        # Get config diff (‘show | compare’ output)
                        diff = conf.diff()
                        # If diff is empty, there is nothing to do
                        if diff is None:
                            print "No change is needed!"
                        # Commit if diff is not empty
                        else:
                            print "Will try to commit this diff on {}:".format(host)
                            print diff
                            conf.commit()
                            print "Committed on the device {}".format(host)
                except ConfigLoadError as error:
                    print "Error: can’t load config on host {} due to {}, skipping it" \
                            .format(host, str(error))
                except LockError:
                    print "Error: can’t lock config on host {}, skipping it".format(host)
                except CommitError as error:
                    print "Error: can’t commit on host {} due to {}, skipping it" \
                            .format(host, str(error))
        except ConnectError as error:
            print "Error: can’t connect to host {} due to {}, skipping it" \
                    .format(host, str(error))


# ==(9)== Entry point =====

# This is a check to make sure this file executes as a main Python script,
# and not imported from another script. If so, run the main() function.
if __name__ == "__main__":
    main()
```

Let’s see what is happening in the script. The following numbered list corresponds to the numbered comments in the script:

1. Import standard PyEZ classes `Device`, `Config` and `exceptions`. Also import a couple of `lxml` library functions and objects that will be used in the script.

2. Define some ‘constant’ data like the name of the apply-macro statement (we use ‘SYNC’), list of destination hosts to synchronize the configuration, and timeout for the PyEZ connection.

3. Define the main() function that will contain the main execution logic. Python does not require you to do so, but it is rather convenient.

4. In this part, you try to import `junos.Junos_Context` object and `jcs.get_secret` functions. Those are only available if you are executing the script directly on a Junos device (on-box). If so, variables `host_from, user_from,` and `passwd_from` are assigned the value of `None`, because in this case, the hostname, username, and password are not required to make a local connection.

If the imports result in an error (Python exception), the `except:` block is executed. In this case, you ask the user for the hostname, username, and password. Also, when working off-box, you will use `getpass.getpass()` function to ask for a password, rather than `jcs.get_secret()`.

5. Read the configuration from a source (HOST_FROM) Junos device. To do this, you use the `<get-configuration>` RPC call using standard Junos PyEZ approach, and store the result to the `conf` variable (it will be of `lxml.etree._Element` type). You also check for possible exceptions and return from the `main()` function, effectively terminating the script execution, if an error is encountered.

6. This is a critical part of the script where you extract only those parts of the configuration that must be synchronized. You first build an `ElementTree` object from your `conf` variable. This is needed so that you can use a `getpath()` method to obtain the full path for any XML element.

Next, you start building the XML configuration (`conf_to_sync` variable) containing all configuration pieces that have `apply-macro` with name given by SYNC_MACRO variable. You search for all such configuration elements using the XPath expression.

You use `replace="replace"` attribute to make sure the loaded part of configuration overrides the existing configuration on the target device for that configuration object or hierarchy. (Follow the comments in a script and the code itself for more details.)

7. Here, you display the resulting configuration to the user and ask for credentials on the destination hosts (HOSTS_TO). If the user does not like the configuration, it is not too late to terminate the script, for example, by pressing ^C at this point.

8. Loop for all hosts in the HOSTS_TO_SYNC list, and lock, load, and commit the configuration using a standard Junos PyEZ approach. Process the possible exceptions.

9. A common practice is to check that a script is executed as a standalone script and not as an imported module. This is true if special `__name__` variable is equal to "`__main__`".

MORE? For a better understanding of this script, you may also want to consult the *Junos Automation Scripting Feature Guide* available at https://www.juniper.net/documentation/en_US/junos/information-products/pathway-pages/config-guide-automation/config-guide-automation.html, Junos PyEZ documentation at https://www.juniper.net/documentation/product/en_US/junos-pyez, and lxml library documentation at https://lxml.de/index.html documentation.

Run the Script

Now that you have developed the script, to run it on the Junos box you have to take the following steps.

1. Copy the `jsync.py` script to the `/var/db/scripts/op/` directory on the Junos device. Use any convenient tool and protocol for that, such as Secure Copy Protocol (SCP). Also, there are some limitations on the Python script file permissions that must be satisfied: the file owner must either be the root or a Junos super-user, and only the file owner is allowed to have write permission for the file.

2. Enter configuration mode and configure Python as a scripting language for Junos:

```
lab@vMX-1> configure
Entering configuration mode

[edit]
lab@vMX-1# set system scripts language python
```

3. Configure `jsync.py` script as an op script and commit:

```
[edit]
lab@vMX-1# set system scripts op file jsync.py

[edit]
lab@vMX-1# commit and-quit
commit complete
Exiting configuration mode

lab@vMX-1>
```

Make sure NETCONF over SSH is enabled on all the destination hosts. If not, to enable NETCONF, enter the following commands:

```
lab@R3> configure
Entering configuration mode

[edit]
lab@R3# set system services netconf ssh

[edit]
lab@R3# commit and-quit
commit complete
Exiting configuration mode
```

Finally, you can run the script from the Junos CLI using the `op` command (see Figure 18.1); it turns out that executing with `| no-more` option is convenient when your script asks for some user input.

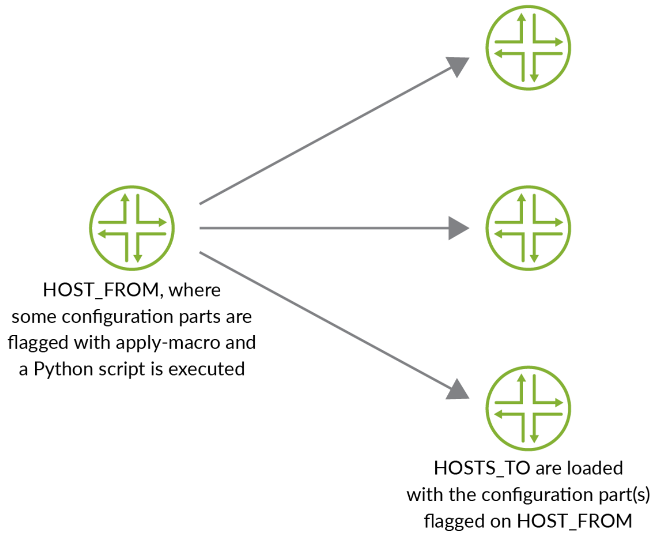

HOST_FROM, where
some configuration parts are
flagged with apply-macro and
a Python script is executed
HOSTS_TO are loaded
with the configuration part(s)
flagged on HOST_FROM

Figure 18.1 Executing the jsync.py Script On-box

```
lab@vMX-1> op jsync | no-more
*** jsync script version 0.1 started ***
*** Working in on-box mode ***
The config to sync is:
<configuration>
<policy-options>
<prefix-list replace="replace">
            <name>ALLOWED_IPS</name>
            <apply-macro>
                <name>SYNC</name>
            </apply-macro>
            <prefix-list-item>
                <name>10.254.0.1/32</name>
            </prefix-list-item>
            <prefix-list-item>
                <name>10.254.0.2/32</name>
            </prefix-list-item>
            <prefix-list-item>
                <name>10.254.0.3/32</name>
            </prefix-list-item>
        </prefix-list>
</policy-options>
</configuration>

Will sync this config to a following set of hosts:
  -   10.254.0.42
  -   10.254.0.35
  -   10.254.0.37
Username on the HOSTS_TO_SYNC: lab
Password on the HOSTS_TO_SYNC: ******

Working on host 10.254.0.42
Will try to commit this diff on 10.254.0.42:

[edit policy-options]
+   prefix-list ALLOWED_IPS {
+       apply-macro SYNC;
+       10.254.0.1/32;
+       10.254.0.2/32;
+       10.254.0.3/32;
+   }
Committed on the device 10.254.0.42

Working on host 10.254.0.35
Will try to commit this diff on 10.254.0.35:

[edit policy-options]
+   prefix-list ALLOWED_IPS {
+       apply-macro SYNC;
+       10.254.0.1/32;
+       10.254.0.2/32;
+       10.254.0.3/32;
+   }
Committed on the device 10.254.0.35

Working on host 10.254.0.37
Will try to commit this diff on 10.254.0.37:

[edit policy-options]
+   prefix-list ALLOWED_IPS {
+       apply-macro SYNC;
+       10.254.0.1/32;
+       10.254.0.2/32;
+       10.254.0.3/32;
+   }
Committed on the device 10.254.0.37
```

As you can see from the script output, the configuration was successfully extracted from a source device and loaded on all three destination devices. This can also be checked on the remote hosts; we omitted the checks here to save space. But note that some of the remote devices are running Junos version 12.1 in the lab setup, and the script still works: the destination devices only need to have NETCONF enabled, and do not need to support on-box Python interpreter (which is available on-box in Junos 16.1 and later).

Now let's modify the prefix list on our HOST_FROM device just a bit:

```
[edit]
lab@vMX-1# set policy-options prefix-list ALLOWED_IPS 10.254.0.4/32

[edit]
lab@vMX-1# commit
commit complete
```

At this point you could run the script from the local Junos device again to load the configuration changes on targets. But let's demonstrate the cross-platform power of Junos PyEZ and run the same script from a Ubuntu Linux box (see Figure 18.2); to be able to do so, you need to have PyEZ installed on your server.

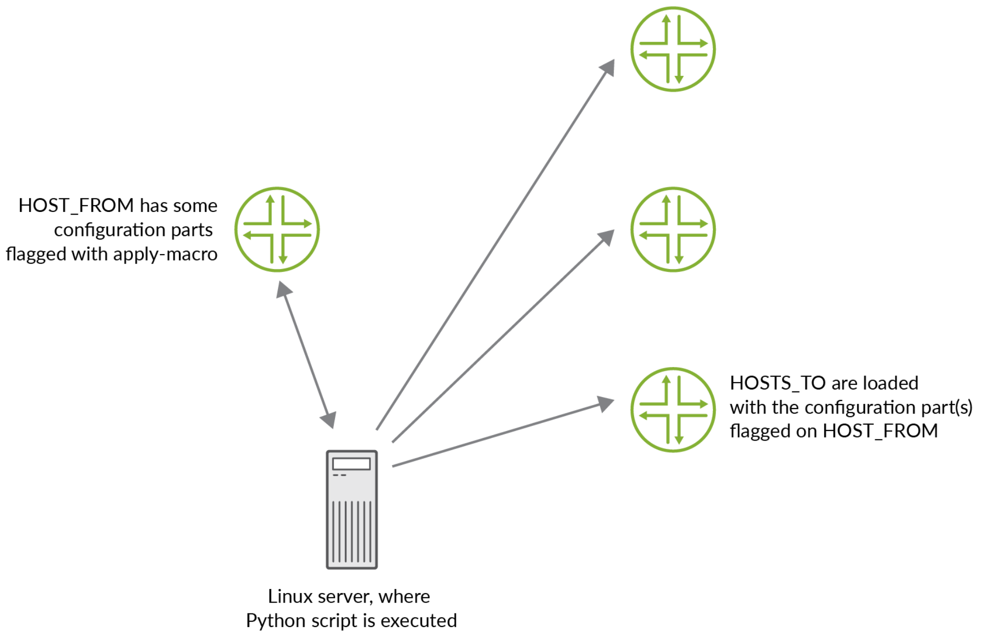

HOST_FROM has some configuration parts flagged with apply-macro
HOSTS_TO are loaded with the configuration part(s) flagged on HOST_FROM
Linux server, where Python script is executed

Figure 18.2 Executing the jsync.py Script Off-box (on a Linux Server)

```
peter@ubuntu16:~$ python2.7 jsync.py
*** jsync script version 0.1 started ***
*** Working in off-box mode ***
Enter host to retrieve configuration (HOST_FROM): 10.254.0.41
Username on the HOST_FROM: lab
Password on the HOST_FROM: ******
The config to sync is:
<configuration>
<policy-options>
<prefix-list replace="replace">
            <name>ALLOWED_IPS</name>
            <apply-macro>
                <name>SYNC</name>
            </apply-macro>
            <prefix-list-item>
                <name>10.254.0.1/32</name>
            </prefix-list-item>
            <prefix-list-item>
                <name>10.254.0.2/32</name>
            </prefix-list-item>
            <prefix-list-item>
                <name>10.254.0.3/32</name>
            </prefix-list-item>
            <prefix-list-item>
                <name>10.254.0.4/32</name>
            </prefix-list-item>
        </prefix-list>
</policy-options>
</configuration>


Will sync this config to a following set of hosts:
  -   10.254.0.42
  -   10.254.0.35
  -   10.254.0.37
Username on the HOSTS_TO_SYNC: lab
Password on the HOSTS_TO_SYNC: ******

Working on host 10.254.0.42
Will try to commit this diff on 10.254.0.42:

[edit policy-options prefix-list ALLOWED_IPS]
+    10.254.0.4/32;

Committed on the device 10.254.0.42

Working on host 10.254.0.35
Will try to commit this diff on 10.254.0.35:

[edit policy-options prefix-list ALLOWED_IPS]
+    10.254.0.4/32;

Committed on the device 10.254.0.35

Working on host 10.254.0.37
Will try to commit this diff on 10.254.0.37:

[edit policy-options prefix-list ALLOWED_IPS]
+    10.254.0.4/32;

Committed on the device 10.254.0.37
```

As you can see, the script worked similarly to how it previously executed on-box, reading and loading configurations properly. The only difference for the user was the need to enter HOST_FROM credentials in the beginning.

Discussion

Among other things, this recipe showed you a somewhat "creative" way of using the `apply-macro` Junos statement. Another possible use for `apply-macro` statements that you may find interesting is to use them as hashtags that can represent a customer, a service, or just about anything meaningful to you. See more details about this idea at the following URL: https://forums.juniper.net/t5/Automation/JUNOS-Hashtags/ta-p/321219.

If you prefer SLAX rather than Python, you can find a SLAX script that is able to copy configuration groups between devices in the Junoscriptorium GitHub repository at https://github.com/Juniper/junoscriptorium/blob/master/library/juniper/op/network/share-data/share-data.slax. The jsync.py Python script shown here can also be used to copy group configurations, but is not limited to that.

The approach that is demonstrated in this recipe is not the only one possible if you have to synchronize some parts of the Junos device configurations. A more common way is to use a centralized configuration management system like Ansible, Salt, or custom Junos PyEZ scripts.

If you are using that approach, you will typically follow these steps:

1. Enter the required configuration on one of the devices first (possibly a lab device), to make sure it works properly.

2. Take that configuration and transform it into a configuration template such as Jinja2 template. Templates may also use variables that change from device to device – in this recipe we assumed configuration to load on all devices is the same.

3. Load the configuration template, together with the required auxiliary files (Ansible playbooks, Salt state files, PyEZ scripts, or whatever you use) onto the management system (server).

4. Instruct the system to load configurations on the specified devices.

You can find the details of how to use this approach in the following *Day One* books:

- For Ansible, *Day One: Automating Junos with Ansible, 2nd Edition*, https://www.juniper.net/uk/en/training/jnbooks/day-one/automation-series/automating-junos-ansible/.
- For Salt, *Day One: Automating Junos with Salt*, https://www.juniper.net/us/en/training/jnbooks/day-one/automating-junos-with-salt/.
- For Junos PyEZ, *Day One: Junos PyEZ Cookbook*, https://www.juniper.net/us/en/training/jnbooks/day-one/automation-series/junos-pyez-cookbook/.

Overall, the template-based approach is more powerful, but it's also a heavier weight. The automation solution shown in this recipe comes as a single, lightweight, on-box script (may also work off-box, as you've seen) and does not require you to manage anything except the Python script file. It copies statements from a device's Junos configuration directly to other devices, which you may sometimes find more convenient.

NOTE The GitHub repository for *this* recipe's script is https://github.com/pklimai/junos-automation-examples/tree/master/jsync. Feel free to send feedback including issues and pull requests via this page.

Recipe 19: Migrating from MC-LAG to ESI-LAG

By Tom Dwyer

- Junos OS Used: 17.4R1.6
- Juniper Platforms General Applicability: QFX, MX, vMX, EX Series

This recipe is about migrating from MC-LAG to ESI-LAG. The Globex Corporation has a small development office that hosts local compute and storage for their Dev and QA environments. They have deployed EVPN in their data centers and want to utilize some of the same features in their development network. They currently have two QFX10ks acting as core switches. In the past they have used MC-LAG for multihoming across the two core switches, but this recipe addresses the steps they need to migrate from MC-LAG to ESI-LAG.

Problem

How to migrate from MC-LAG to ESI-LAG.

Solution

Globex is familiar with EVPN as they have already deployed it at their two main data centers. In those data centers they have a spine-and-leaf topology and are using EVPN for Data Center Interconnect (DCI). The development office would like to take advantage of some of the EVPN features including ESI-LAG.   They want to start testing N-way multihoming with their compute environment. The initial deployment will be a spine-only architecture that will allow them to use these EVPN features without having to completely redesign their network (they may add a third core switch in the future).

The current network has two cores, CORE-1 and CORE-2, and a switch for the developers' computers wired connectivity (IDF-1). The core switches have MC-LAG configured southbound to provide connectivity and redundancy without having to worry about Spanning Tree Protocol. They are using VRRP for gateway redundancy for their server VLAN.

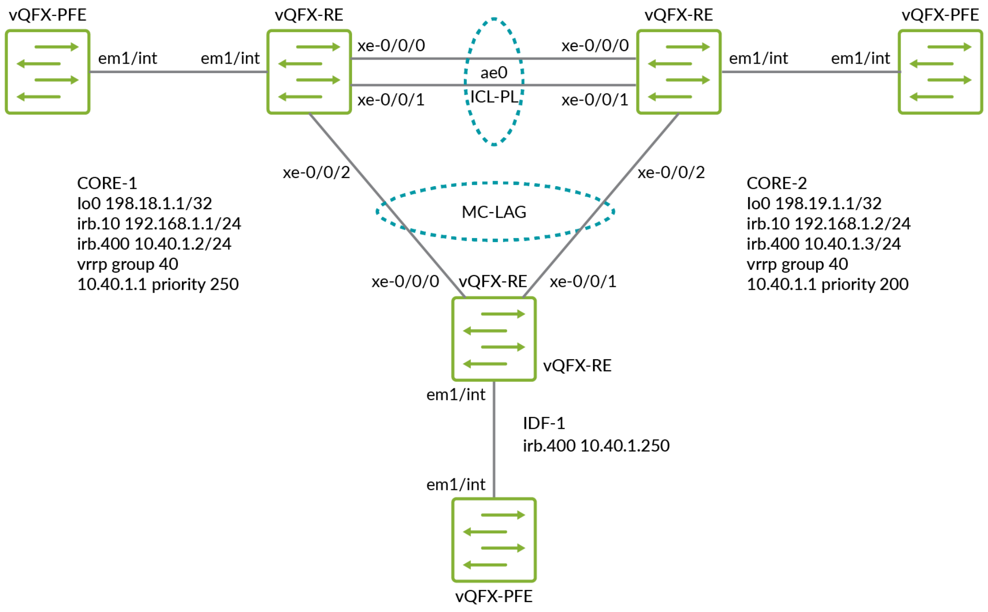


Figure 19.1 Current MC-LAG Configuration

The conversion process will move Globex from an active/passive Layer 3 configuration with Virtual Router Redundancy Protocol (VRRP) to an active/active configuration with EVPN virtual gateways. We will use a static MAC address 00:de:ad:be:ef:11 for our virtual gateways instead of the system defined MAC 00:00:5e:00:01:01, which is also used for VRRP. Using a statically defined MAC will help us during our testing and troubleshooting phase as it's easily identifiable. The system commits should happen at the end of the recipe to lessen the impact of service disruption.

The first step is to start the building blocks for EVPN. The first thing you need to set up is the underlay network to allow the VTEPs to have connectivity to each other. Let's use OSPF to provide this underlay functionality:

```
root@CORE-1#set interfaces lo0 unit 0 family inet address 198.18.1.1/32
root@CORE-1#set routing-options router-id 198.18.1.1
root@CORE-1#set protocols ospf area 0.0.0.0 interface irb.10
root@CORE-1#set protocols ospf area 0.0.0.0 interface lo0.0 passive

root@CORE-2#set interfaces lo0 unit 0 family inet address 198.19.1.1/32
root@CORE-2#set routing-options router-id 198.19.1.1
root@CORE-2#set protocols ospf area 0.0.0.0 interface irb.10
root@CORE-2#set protocols ospf area 0.0.0.0 interface lo0.0 passive

root@CORE-1> show ospf neighbor
Address          Interface              State     ID               Pri  Dead
192.168.1.2      irb.10                 Full      198.19.1.1       128    39

root@CORE-1> show route protocol ospf | match 198
198.19.1.1/32      *[OSPF/10] 12:51:30, metric 1
```

```
root@CORE-2> show ospf neighbor
Address          Interface              State     ID               Pri  Dead
192.168.1.1      irb.10                 Full      198.18.1.1       128   34

{master:0}
root@CORE-2> show route protocol ospf | match 198
198.18.1.1/32      *[OSPF/10] 12:52:25, metric 1
```

Now that the easy part is done, we get to complete the interesting steps: there is configuration syntax that will conflict from the old MC-LAG to the new EVPN configuration. In a production network you would shut down both sides of the MC-LAG facing the IDF switch to not risk any suboptimal behavior. The beauty of Junos is that you could stage the entire next section and commit to strip and replace MC-LAG with ESI-LAG in one fell swoop. Some will lean toward a more modest approach and your risk tolerance may vary. Here, we'll shut down the MC-LAG southbound AEs on both core switches:

```
root@CORE-1> configure
Entering configuration mode

{master:0}[edit]
root@CORE-1# set interfaces ae1 disable
{master:0}[edit]
root@CORE-1# commit and-quit
configuration check succeeds
commit complete
Exiting configuration mode

root@CORE-2> configure
Entering configuration mode

{master:0}[edit]
root@CORE-2# set interfaces ae1 disable
{master:0}[edit]
root@CORE-2# commit and-quit
configuration check succeeds
commit complete
Exiting configuration mode

root@IDF-1> show interfaces ae0 terse
Interface              Admin Link Proto    Local                 Remote
ae0                    up    down
ae0.0                  up    down eth-switch

{master:0}
root@IDF-1> show lacp interfaces
Aggregated interface: ae0
<omitted>
    LACP protocol:        Receive State  Transmit State          Mux State
      xe-0/0/0                Defaulted   Fast periodic           Detached
      xe-0/0/1                Defaulted   Fast periodic           Detached
```

Now that the southbound AEs are shut down you can move toward the heavy lifting of migrating over to ESI-LAG.

The next step is to remove `iccp` and `multi-chassis-protection`:

```
root@CORE-1>configure
root@CORE-1#delete protocols iccp local-ip-addr 192.168.1.1
root@CORE-1#delete protocols iccp peer 192.168.1.2 redundancy-group-id-list 1
root@CORE-1#delete protocols iccp peer 192.168.1.2 liveness-detection minimum-receive-interval 3000
root@CORE-1#delete protocols iccp peer 192.168.1.2 liveness-detection transmit-interval minimum-
interval 3000
root@CORE-1#delete multi-chassis multi-chassis-protection 192.168.1.2 interface ae0
root@CORE-1#delete switch-options service-id 1

root@CORE-2>configure
root@CORE-2#delete protocols iccp local-ip-addr 192.168.1.2
root@CORE-2#delete protocols iccp peer 192.168.1.1 redundancy-group-id-list 1
root@CORE-2#delete protocols iccp peer 192.168.1.1 liveness-detection minimum-receive-interval 3000
root@CORE-2#delete protocols iccp peer 192.168.1.1 liveness-detection transmit-interval minimum-
interval 3000
root@CORE-2#delete multi-chassis multi-chassis-protection 192.168.1.1 interface ae0
root@CORE-2#delete switch-options service-id 1
```

Next remove the `mc-ae` configuration from the MC-LAG interface. Leave the `lacp system-id` because you will reuse that in your ESI-LAG:

```
root@CORE-2#delete switch-options service-id 1
root@CORE-1#delete interfaces ae1 aggregated-ether-options lacp admin-key 1
root@CORE-1#delete interfaces ae1 aggregated-ether-options mc-ae mc-ae-id 1
root@CORE-1#delete interfaces ae1 aggregated-ether-options mc-ae chassis-id 0
root@CORE-1#delete interfaces ae1 aggregated-ether-options mc-ae mode active-active
root@CORE-1#delete interfaces ae1 aggregated-ether-options mc-ae status-control active
root@CORE-1#delete interfaces ae1 aggregated-ether-options mc-ae init-delay-time 200
root@CORE-1#delete interfaces ae1 aggregated-ether-options mc-ae redundancy-group 1

root@CORE-2#delete interfaces ae1 aggregated-ether-options lacp admin-key 1
root@CORE-2#delete interfaces ae1 aggregated-ether-options mc-ae mc-ae-id 1
root@CORE-2#delete interfaces ae1 aggregated-ether-options mc-ae chassis-id 1
root@CORE-2#delete interfaces ae1 aggregated-ether-options mc-ae mode active-active
root@CORE-2#delete interfaces ae1 aggregated-ether-options mc-ae status-control standby
root@CORE-2#delete interfaces ae1 aggregated-ether-options mc-ae init-delay-time 200
root@CORE-2#delete interfaces ae1 aggregated-ether-options mc-ae redundancy-group 1
```

The ae0 interface will be repurposed from the ICL-PL link to transport only the vlan for the EVPN underlay. Remove VLAN 400 from this link because you will use EVPN for transport of this VLAN.   This concept of the directly connected spines might look odd at first (see Figure 19.2). Remember the underlying topology is about providing the VTEP’s connectivity. In a normal spine and leaf fabric, the spines are not directly connected and the spines connect to each other via the leaf switches.

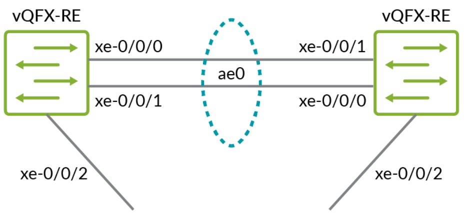


Figure 19.2 ICL-PL Link Reconfigured for EVPN Underlay

```
root@CORE-1# set interfaces ae0 description "EVPN Spine-Only"
root@CORE-1# delete interface ae0 unit 0 family ethernet-switching vlan members 400
root@CORE-2# set interfaces ae0 description "EVPN Spine-Only"
root@CORE-2# delete interface ae0 unit 0 family ethernet-switching vlan members 400
root@CORE-1#rename vlans ICL-PL to UNDERLAY
root@CORE-2#rename vlans ICL-PL to UNDERLAY
```

Now let's build out the EVPN configuration. First the overlay:

CORE-1

```
set routing-options autonomous-system 65000
set protocols bgp log-updown
set protocols bgp group EVPN-OVERLAY type internal
set protocols bgp group EVPN-OVERLAY local-address 198.18.1.1
set protocols bgp group EVPN-OVERLAY family evpn signaling
set protocols bgp group EVPN-OVERLAY neighbor 198.19.1.1
```

CORE-2

```
set routing-options autonomous-system 65000
set protocols bgp log-updown
set protocols bgp group EVPN-OVERLAY type internal
set protocols bgp group EVPN-OVERLAY local-address 198.19.1.1
set protocols bgp group EVPN-OVERLAY family evpn signaling
set protocols bgp group EVPN-OVERLAY neighbor 198.18.1.1
```

And now define your `evpn` protocol options:

BOTH CORE SW's

```
set protocols evpn vni-options vni 400 vrf-target target:1:400
set protocols evpn encapsulation vxlan
set protocols evpn multicast-mode ingress-replication
set protocols evpn extended-vni-list 400
set vlans CORE vxlan vni 400
```

You will need to define an import policy to match the route targets for the type 1 and type 2 EVPN routes:

```
set policy-options community CORE members target:1:400
set policy-options community ESI-LAG members target:65000:1
set policy-options policy-statement EVPN-IMPORT term 1 from community CORE
set policy-options policy-statement EVPN-IMPORT term 1 then accept
set policy-options policy-statement EVPN-IMPORT term 2 from community ESI-LAG
set policy-options policy-statement EVPN-IMPORT term 2 then accept
set policy-options policy-statement EVPN-IMPORT term 3 then reject
```

Next define the route distinguisher and the `vrf-import` policy to install the routes:

```
root@CORE-1#set switch-options vtep-source-interface lo0.0
root@CORE-1#set switch-options route-distinguisher 198.18.1.1:1
root@CORE-1#set switch-options vrf-import EVPN-IMPORT
root@CORE-1#set switch-options vrf-target target:65000:1

root@CORE-2#set switch-options vtep-source-interface lo0.0
root@CORE-2#set switch-options route-distinguisher 198.19.1.1:1
root@CORE-2#set switch-options vrf-import EVPN-IMPORT
root@CORE-2#set switch-options vrf-target target:65000:1
```

Replace the VRRP configuration that was used in MC-LAG with the virtual gateway address (VGA). With VRRP, one of the spines is used for the gateway with VGA; we can use both spines to provide gateway support. This is definitely a benefit to using EVPN and ESI-LAG over MC-LAG:

```
root@CORE-1#delete interfaces irb unit 400 family inet address 10.40.1.2/24 vrrp-group 40
root@CORE-2#delete interfaces irb unit 400 family inet address 10.40.1.3/24 vrrp-group 40
root@CORE-1#set interfaces irb unit 400 family inet address 10.40.1.2/24 virtual-gateway-address
10.40.1.1
root@CORE-1#set interfaces irb unit 400 virtual-gateway-v4-mac 00:de:ad:be:ef:11
root@CORE-2#set interfaces irb unit 400 family inet address 10.40.1.3/24 virtual-gateway-address
10.40.1.1
root@CORE-2#set interfaces irb unit 400 virtual-gateway-v4-mac 00:de:ad:be:ef:11
```

The next step is to configure the manual 10 octet ESI value for multihoming. The first octet needs to be configured as 00. The remaining nine are user defined (see Figure 19.3). It is important to understand, however, that the first six significant octets after the first octet are used as part of the `es-import-target` value.

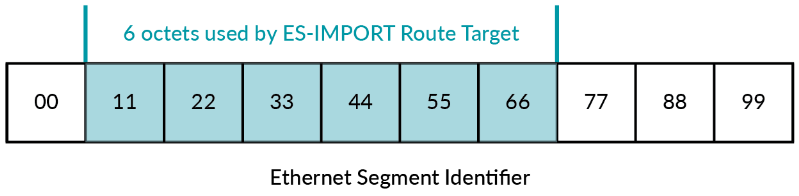

6 octets used by ES-IMPORT Route Target
00
11
22
33
44
55
66
77
88
99
Ethernet Segment Identifier

Figure 19.3 Ethernet Segment Identifier Significant Octects for ESI-import

```
set interfaces ae1 esi 00:11:22:33:44:55:66:77:88:99
set interfaces ae1 esi all-active
```

Well, it looks like things are ready to commit and to bring up the southbound aggregated ethernet interfaces:

```
{master:0}[edit interfaces ae1]
root@CORE-1# delete disable
{master:0}[edit interfaces ae1]
root@CORE-2# delete disable

root@CORE-1# commit and-quit
configuration check succeeds
commit complete
Exiting configuration mode

root@CORE-2# commit and-quit
configuration check succeeds
commit complete
Exiting configuration mode
```

Validation

Let’s see that BGP is up and there are EVPN routes present:

```
root@CORE-1> show bgp summary
Groups: 1 Peers: 1 Down peers: 0
Table          Tot Paths  Act Paths Suppressed    History Damp State    Pending
bgp.evpn.0
                      10         10          0          0          0          0
Peer                     AS      InPkt     OutPkt    OutQ   Flaps Last Up/Dwn State|#
Active/Received/Accepted/Damped...
198.19.1.1            65000       2754       2688       0       0    19:48:38 Establ
  bgp.evpn.0: 10/10/10/0
  default-switch.evpn.0: 9/9/9/0
  __default_evpn__.evpn.0: 1/1/1/0

root@CORE-1> show route table default-switch.evpn.0

default-switch.evpn.0: 22 destinations, 22 routes (22 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

1:198.19.1.1:0::112233445566778899::FFFF:FFFF/192 AD/ESI
                   *[BGP/170] 00:25:31, localpref 100, from 198.19.1.1
                      AS path: I, validation-state: unverified
                    > to 192.168.1.2 via irb.10
1:198.19.1.1:0::050000fde80000019000::FFFF:FFFF/192 AD/ESI
                   *[BGP/170] 19:38:16, localpref 100, from 198.19.1.1
                      AS path: I, validation-state: unverified
                    > to 192.168.1.2 via irb.10
1:198.18.1.1:1::112233445566778899::0/192 AD/EVI
                   *[EVPN/170] 00:25:37
                      Indirect
1:198.19.1.1:1::112233445566778899::0/192 AD/EVI
                   *[BGP/170] 00:25:32, localpref 100, from 198.19.1.1
                      AS path: I, validation-state: unverified
                    > to 192.168.1.2 via irb.10
2:198.18.1.1:1::400::00:50:79:66:68:08/304 MAC/IP
                   *[EVPN/170] 00:02:33
                      Indirect
2:198.18.1.1:1::400::00:de:ad:be:ef:11/304 MAC/IP
                   *[EVPN/170] 19:38:19
                      Indirect
```

```
2:198.18.1.1:1::400::02:05:86:71:96:00/304 MAC/IP
                   *[EVPN/170] 00:08:38
                      Indirect
2:198.18.1.1:1::400::02:05:86:71:a7:00/304 MAC/IP
                   *[EVPN/170] 19:38:19
                      Indirect
2:198.19.1.1:1::400::00:50:79:66:68:07/304 MAC/IP
                   *[BGP/170] 19:26:23, localpref 100, from 198.19.1.1
                      AS path: I, validation-state: unverified
                    > to 192.168.1.2 via irb.10
```

Remember when we created the ESI value, and how we talked about the importance of the first six significant octets? Here is an auto-generated `es-import` policy based on a target value:

```
root@CORE-1> show policy __vrf-import-__default_evpn__-internal__
Policy __vrf-import-__default_evpn__-internal__:
    Term unnamed:
        from community __vrf-community-__default_evpn__-import-internal__ [es-import-
target:11-22-33-44-55-66 ]
        then accept
    Term unnamed:
        then reject


root@CORE-1> show evpn instance extensive
Instance: __default_evpn__
<omitted>
  MAC database status                     Local  Remote
    MAC advertisements:                       1       3
    MAC+IP advertisements:                    2       2
    Default gateway MAC advertisements:       2       1
  Number of local interfaces: 2 (2 up)
    Interface name  ESI                            Mode             Status     AC-Rol
e
    ae0.0           00:00:00:00:00:00:00:00:00:00  single-homed     Up         Root
    ae1.0           00:11:22:33:44:55:66:77:88:99  all-active       Up         Root
  Number of IRB interfaces: 1 (1 up)
    Interface name  VLAN   VNI    Status  L3 context
    irb.400                400    Up      master
  Number of protect interfaces: 0
  Number of bridge domains: 1
    VLAN  Domain ID   Intfs / up    IRB intf   Mode             MAC sync  IM route la
bel  SG sync  IM core nexthop
    400   400            1    1     irb.400    Extended         Enabled   400
     Disabled
  Number of neighbors: 1
    Address               MAC    MAC+IP        AD        IM        ES Leaf-label
    198.19.1.1              3         2         3         1         0
  Number of ethernet segments: 2
    ESI: 00:11:22:33:44:55:66:77:88:99
      Status: Resolved by IFL ae1.0
      Local interface: ae1.0, Status: Up/Forwarding
      Number of remote PEs connected: 1
        Remote PE        MAC label  Aliasing label  Mode
        198.19.1.1       0          0               all-active
      DF Election Algorithm: MOD based
      Designated forwarder: 198.18.1.1
      Backup forwarder: 198.19.1.1
      Last designated forwarder update: Mar 05 03:41:36

root@CORE-1> show ethernet-switching table

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
  CORE              00:50:79:66:68:07   D       vtep.32769           198.19.1.1
  CORE              00:de:ad:be:ef:11   DR      esi.1730             05:00:00:fd:e8:00:00:01:90:00
  CORE              02:05:86:71:7c:00   D       vtep.32769           198.19.1.1
  UNDERLAY           02:05:86:71:7c:00   D       ae0.0

root@CORE-2> show ethernet-switching table

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


Ethernet switching table : 4 entries, 4 learned
Routing instance : default-switch
   Vlan                MAC                 MAC      Logical                A
ctive
   name                address             flags    interface              s
ource
   CORE                00:50:79:66:68:07   D        xe-0/0/3.0
   CORE                00:de:ad:be:ef:11   DR       esi.1766               0
5:00:00:fd:e8:00:00:01:90:00
   CORE                02:05:86:71:a7:00   D        vtep.32769             1
98.18.1.1
   UNDERLAY            02:05:86:71:a7:00   D        ae0.0
```

Validate that both sides of the aggregated Ethernet interface are up on the IDF switch and are participating in LACP. You'll see the defined virtual gateway MAC being learned in the Ethernet switching table, too:

```
root@IDF-1> show lacp interfaces ae0
Aggregated interface: ae0
    LACP state:       Role   Exp   Def  Dist  Col  Syn  Aggr  Timeout  Activity
      xe-0/0/0       Actor    No    No   Yes  Yes  Yes   Yes     Fast    Active
      xe-0/0/0     Partner    No    No   Yes  Yes  Yes   Yes     Fast   Passive
      xe-0/0/1       Actor    No    No   Yes  Yes  Yes   Yes     Fast    Active
      xe-0/0/1     Partner    No    No   Yes  Yes  Yes   Yes     Fast   Passive
    LACP protocol:        Receive State  Transmit State          Mux State
      xe-0/0/0                  Current   Fast periodic Collecting distributing
      xe-0/0/1                  Current   Fast periodic Collecting distributing

root@IDF-1> show ethernet-switching table
```

```
MAC flags (S - static MAC, D - dynamic MAC, L - locally learned, P - Persist
ent static, C - Control MAC
           SE - statistics enabled, NM - non configured MAC, R - remote PE M
AC, O - ovsdb MAC)


Ethernet switching table : 2 entries, 2 learned
Routing instance : default-switch
    Vlan                MAC                 MAC         Age    Logical
          NH         RTR
    name                address             flags              interface
          Index      ID
    CORE                00:50:79:66:68:08   D             -   xe-0/0/2.0
         0          0
    CORE                00:de:ad:be:ef:11   D             -   ae0.0
```

A PC has been added for validation and testing and it is connected to the IDF switch and one that is connected to CORE-2. Layer 2 and Layer 3 connectivity have been verified. The MC-LAG to ESI-LAG conversion is over.

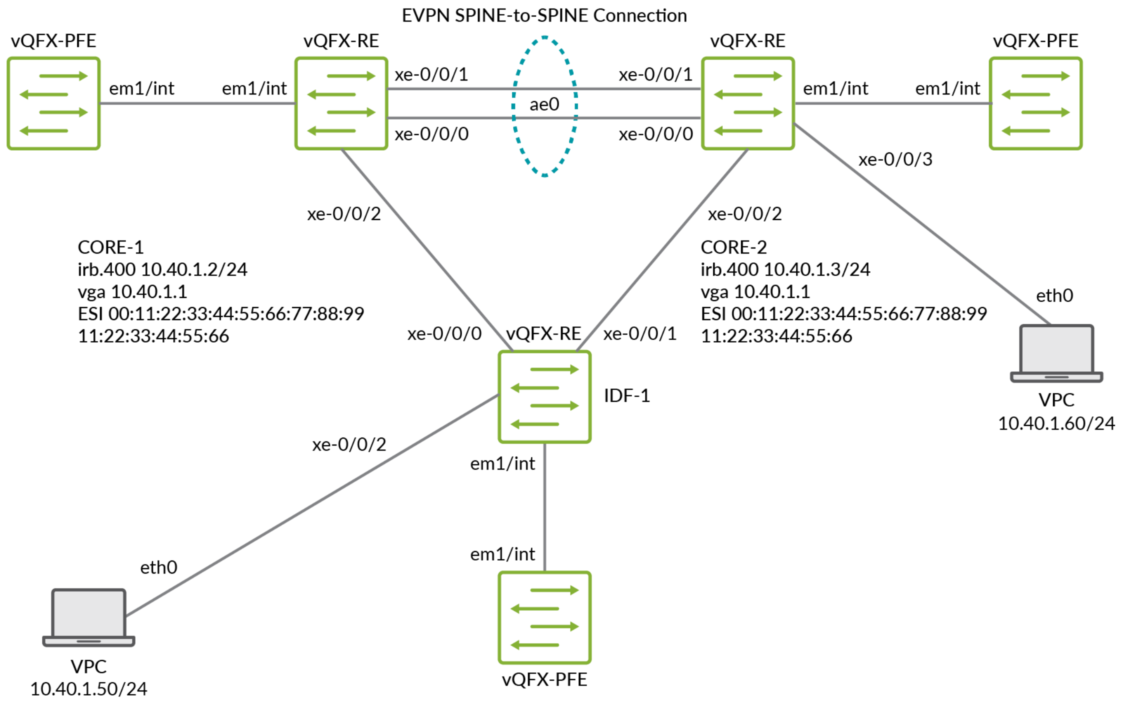


Figure 19.4 The Final MC-LAG to ESI-LAG Conversion

```
root@CORE-1> show arp
MAC Address       Address         Name                      Interface               Flags
00:de:ad:be:ef:11 10.40.1.1       10.40.1.1                 irb.400                 permanent
published gateway
02:05:86:71:7c:00 10.40.1.3       10.40.1.3                 irb.400 [vtep.32769]    permanent remote
00:50:79:66:68:08 10.40.1.50      10.40.1.50                irb.400 [ae1.0]         permanent remote
00:50:79:66:68:07 10.40.1.60      10.40.1.60                irb.400 [vtep.32769]    permanent remote
02:05:86:71:96:00 10.40.1.254     10.40.1.254               irb.400 [ae1.0]         permanent remote
50:00:00:04:00:01 169.254.0.1     169.254.0.1               em1.0                   none
02:05:86:71:7c:00 192.168.1.2     192.168.1.2               irb.10 [ae0.0]          none
Total entries: 7
```

Discussion

One of the takeaways from this solution is showing you that you can get the benefits of EVPN without having to deploy a complete spine-and-leaf fabric. The other takeaway is to remember to review the RFC (RFC 7432) documentation as it can provide some details that are commonly overlooked when trying to understand how a technology should be deployed. One thing that commonly gets overlooked is how the ESI octet significance works with the hidden es-import policy. Not understanding this can create situations where the ESI values are not unique to this import policy which can add load to other switches in your network.

This solution was also completely built using a virtual lab environment using eve-ng with the vQFX images. We are not all lucky enough to have a full physical lab, or virtual labs, to allow us to step through the process ahead of time and under control, learning along the way and getting the process straight. Being able to lab this up ahead of time on the vQFX helps to get the steps straight. The adage of measure twice and cut once is something that engineers should try to live by.

This recipe was not meant to be an all-encompassing expose on EVPN – it is a recipe that migrates off of MC-LAG, and the EVPN tasks were the basics to get it up so we could utilize ESI-LAG under the hood. There are a ton of other knobs and optimizations that could be added to this as a Phase 2 approach. Get in the lab and investigate!