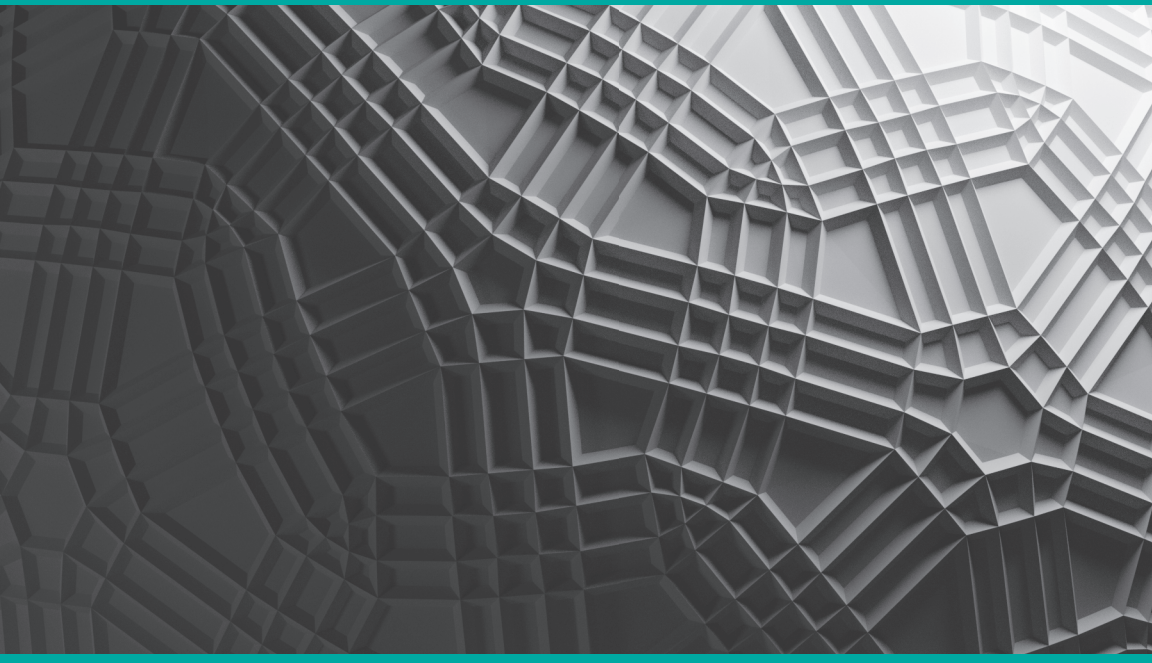
O'REILLY®

BGP in the Data Center



Dinesh G. Dutt

BGP in the Data Center

Dinesh G. Dutt

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

BGP in the Data Center
by Dinesh G. Dutt

Printed in the United States of America.

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

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

Editors: Courtney Allen and Virginia Wilson
Production Editor: Kristen Brown
Copyeditor: Octal Publishing, Inc.
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest
Tech Reviewers: Cody Bunch and Akhil Behl

June 2017: First Edition

Revision History for the First Edition
2017-06-19: First Release
2017-07-28: Second Release

978-1-491-98340-9

[LSI]

Table of Contents

Preface

This little booklet is the outcome of the questions I've frequently encountered in my engagement with various customers, big and small, in their journey to build a modern data center.

BGP in the data center is a rather strange beast, a little like the title of that Sting song, "An Englishman in New York." While its entry into the data center was rather unexpected, it has swiftly asserted itself as the routing protocol of choice in data center deployments.

Given the limited scope of a booklet like this, the goals of the book and the assumptions about the audience are critical. The book is designed for network operators and engineers who are conversant in networking and the basic rudiments of BGP, and who want to understand how to deploy BGP in the data center. I do not expect any advanced knowledge of BGP's workings or experience with any specific router platform.

The primary goal of this book is to gather in a single place the theory and practice of deploying BGP in the data center. I cover the design and effects of a Clos topology on network operations before moving on to discuss how to adapt BGP to the data center. Two chapters follow where we'll build out a sample configuration for a two-tier Clos network. The aim of this configuration is to be simple and automatable. We break new ground in these chapters with ideas such as BGP unnumbered. The book finishes with a discussion of deploying BGP on servers in order to deal with the buildout of microservices applications and virtual firewall and load balancer services. Although I do not cover the actual automation playbooks in this book, the accompanying software on GitHub will provide a virtual network on a sturdy laptop for you to play with.

The people who really paid the price, as I took on the writing of this booklet along with my myriad other tasks, were my wife Shanthala and daughter Maya. Thank you. And it has been nothing but a pleasure and a privilege to work with Cumulus Networks' engineering, especially the routing team, in developing and working through ideas to make BGP simpler to configure and manage.

Software Used in This Book

There are many routing suites available today, some vendor-proprietary and others open source. I've picked the open source FRRouting routing suite as the basis for my configuration samples. It implements many of the innovations discussed in this book. Fortunately, its configuration language mimics that of many other traditional vendor routing suites, so you can translate the configuration snippets easily into other implementations.

The automation examples listed on the GitHub page all use Ansible and Vagrant. Ansible is a popular, open source server automation tool that is very popular with network operators due to its simple, no-programming-required model. Vagrant is a popular open source tool used to spin up networks on a laptop using VM images of router software.

CHAPTER 1

Introduction to Data Center Networks

A network exists to serve the connectivity requirements of applications, and applications serve the business needs of their organization. As a network designer or operator, therefore, it is imperative to first understand the needs of the modern data center, and the network topology that has been adapted for the data centers. This is where our journey begins. My goal is for you to understand, by the end of the chapter, the network design of a modern data center network, given the applications' needs and the scale of the operation.

Data centers are much bigger than they were a decade ago, with application requirements vastly different from the traditional client–server applications, and with deployment speeds that are in seconds instead of days. This changes how networks are designed and deployed.

The most common routing protocol used inside the data center is Border Gateway Protocol (BGP). BGP has been known for decades for helping internet-connected systems around the world find one another. However, it is useful within a single data center, as well. BGP is standards-based and supported by many free and open source software packages.

It is natural to begin the journey of deploying BGP in the data center with the design of modern data center networks. This chapter is an answer to questions such as the following:

- What are the goals behind a modern data center network design?
- How are these goals different from other networks such as enterprise and campus?
- Why choose BGP as the routing protocol to run the data center?

Requirements of a Data Center Network

Modern data centers evolved primarily from the requirements of web-scale pioneers such as Google and Amazon. The applications that these organizations built—primarily search and cloud—represent the third wave of application architectures. The first two waves were the monolithic single-machine applications, and the client–server architecture that dominated the landscape at the end of the past century.

The three primary characteristics of this third-wave of applications are as follows:

Increased server-to-server communication
: Unlike client–server architectures, the modern data center applications involve a lot of server-to-server communication. Client–server architectures involved clients communicating with fairly monolithic servers, which either handled the request entirely by themselves, or communicated in turn to at most a handful of other servers such as database servers. In contrast, an application such as search (or its more popular incarnation, Hadoop), can employ tens or hundreds of mapper nodes and tens of reducer nodes. In a cloud, a customer's virtual machines (VMs) might reside across the network on multiple nodes but need to communicate seamlessly. The reasons for this are varied, from deploying VMs on servers with the least load to scaling-out server load, to load balancing. A microservices architecture is another example in which there is increased server-to-server communication. In this architecture, a single function is decomposed into smaller building blocks that communicate together to achieve the final result. The promise of such an architecture is that each block can therefore be used in multiple applications, and each block can be enhanced, modified, and fixed more easily and independently from the other

blocks. Server-to-server communications is often called *East-West traffic*, because diagrams typically portray servers side-by-side. In contrast, traffic exchanged between local networks and external networks is called *North-South traffic*.

Scale
: If there is one image that evokes a modern data center, it is the sheer scale: rows upon rows of dark, humming, blinking machines in a vast room. Instead of a few hundred servers that represented a large network in the past, modern data centers range from a few hundred to a hundred thousand servers in a single physical location. Combined with increased server-to-server communication, the connectivity requirements at such scales force a rethink of how such networks are constructed.

Resilience
: Unlike the older architectures that relied on a reliable network, modern data center applications are designed to work in the presence of failures—nay, they assume failures as a given. The primary aim is to limit the effect of a failure to as small a footprint as possible. In other words, the "blast radius" of a failure must be constrained. The goal is an end-user experience mostly unaffected by network or server failures.

Any modern data center network has to satisfy these three basic application requirements. Multitenant networks such as public or private clouds have an additional consideration: rapid deployment and teardown of a virtual network. Given how quickly VMs—and now containers—can spin up and tear down, and how easily a customer can spin up a new private network in the cloud, the need for rapid deployment becomes obvious.

The traditional network design scaled to support more devices by deploying larger switches (and routers). This is the *scale-in* model of scaling. But these large switches are expensive and mostly designed to support only a two-way redundancy. The software that drives these large switches is complex and thus prone to more failures than simple, fixed-form factor switches. And the scale-in model can scale only so far. No switch is too large to fail. So, when these larger switches fail, their blast radius is fairly large. Because failures can be disruptive if not catastrophic, the software powering these "god-boxes" try to reduce the chances of failure by adding yet more complexity; thus they counterproductively become more prone to failure

as a result. And due to the increased complexity of software in these boxes, changes must be slow to avoid introducing bugs into hardware or software.

Rejecting this paradigm that was so unsatisfactory in terms of reliability and cost, the web-scale pioneers chose a different network topology to build their networks.

Clos Network Topology

The web-scale pioneers picked a network topology called *Clos* to fashion their data centers. Clos networks are named after their inventor, Charles Clos, a telephony networking engineer, who, in the 1950s, was trying to solve a problem similar to the one faced by the web-scale pioneers: how to deal with the explosive growth of telephone networks. What he came up with we now call the Clos network topology or architecture.

Figure 1-1 shows a Clos network in its simplest form. In the diagram, the green nodes represent the switches and the gray nodes the servers. Among the green nodes, the ones at the top are *spine nodes*, and the lower ones are *leaf nodes*. The spine nodes connect the leaf nodes with one another, whereas the leaf nodes are how servers connect to the network. Every leaf is connected to every spine node, and, obviously, vice versa. C'est tout!

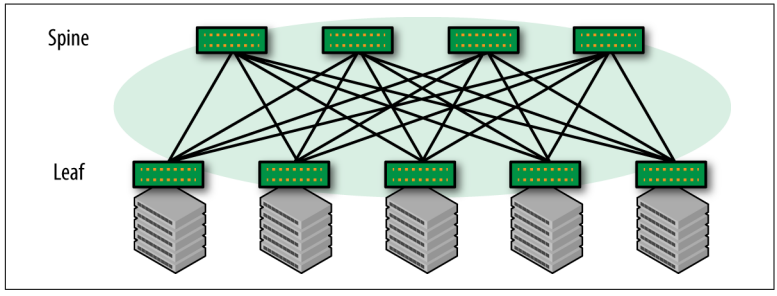


Figure 1-1. A simple two-tier Clos network

Let's examine this design in a little more detail. The first thing to note is the uniformity of connectivity: servers are typically three network hops away from any other server. Next, the nodes are quite homogeneous: the servers look alike, as do the switches. As required by the modern data center applications, the connectivity matrix is quite rich, which allows it to deal gracefully with failures. Because

there are so many links between one server and another, a single failure, or even multiple link failures, do not result in complete connectivity loss. Any link failure results only in a fractional loss of bandwidth as opposed to a much larger, typically 50 percent, loss that is common in older network architectures with two-way redundancy.

The other consequence of having many links is that the bandwidth between any two nodes is quite substantial. The bandwidth between nodes can be increased by adding more spines (limited by the capacity of the switch).

We round out our observations by noting that the endpoints are all connected to leaves, and that the spines merely act as connectors. In this model, the functionality is pushed out to the edges rather than pulled into the spines. This model of scaling is called a *scale-out* model.

You can easily determine the number of servers that you can connect in such a network, because the topology lends itself to some simple math. If we want a nonblocking architecture—i.e., one in which there's as much capacity going between the leaves and the spines as there is between the leaves and the servers—the total number of servers that can be connected is $n^2 / 2$, where n is the number of ports in a switch. For example, for a 64-port switch, the number of servers that you can connect is 64 * 64 / 2 = 2,048 servers. For a 128-port switch, the number of servers jumps to 128 * 128 / 2 = 8,192 servers. The general equation for the number of servers that can be connected in a simple leaf-spine network is $n * m / 2$, where n is the number of ports on a leaf switch, and m is the number of ports on a spine switch.

In reality, servers are interconnected to the leaf via lower-speed links and the switches are interconnected by higher-speed links. A common deployment is to interconnect servers to leaves via 10 Gbps links, while interconnecting switches with one another via 40 Gbps links. Given the rise of 100 Gbps links, an up-and-coming deployment is to use 25 Gbps links to interconnect servers to leaves, and 100 Gbps links to interconnect the switches.

Due to power restrictions, most networks have at most 40 servers in a single rack (though new server designs are pushing this limit). At the time of this writing, the most common higher-link speed switches have at most 32 ports (each port being either 40 Gbps or

100 Gbps). Thus, the maximum number of servers that you can pragmatically connect with a simple leaf–spine network is 40 * 32 = 1,280 servers. However, 64-port and 128-port versions are expected soon.

Although 1,280 servers is large enough for most small to middle enterprises, how does this design get us to the much-touted tens of thousands or hundreds of thousands of servers?

Three-Tier Clos Networks

Figure 1-2 depicts a step toward solving the scale-out problem defined in the previous section. This is what is called a *three-tier Clos network*. It is just a bunch of leaf–spine networks—or two-tier Clos networks—connected by another layer of spine switches. Each two-tier network is called a *pod* or *cluster*, and the third tier of spines connecting all the pods is called an *interpod spine* or *intercluster spine layer*. Quite often, the first tier of switches, the ones servers connect to, are called *top-of-rack* (ToR) because they're typically placed at the top of each rack; the next tier of switches, are called leaves, and the final tier of switches, the ones connecting the pods, are called spines.

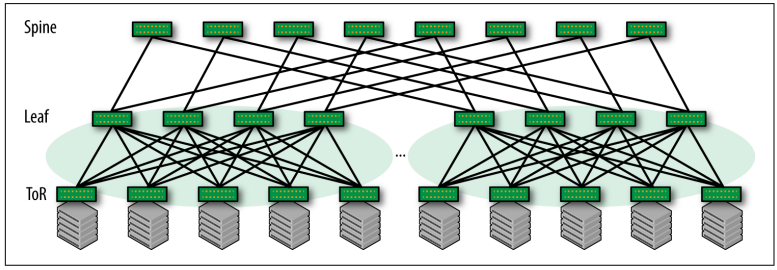


Figure 1-2. Three-tier Clos network

In such a network, assuming that the same switches are used at every tier, the total number of servers that you can connect is $n^3 / 4$. Assuming 64-port switches, for example, we get $64^3 / 4 = 65{,}536$ servers. Assuming the more realistic switch port numbers and servers per rack from the previous section, we can build 40 * 16 * 16 = 10,240 servers.

Large-scale network operators overcome these port-based limitations in one of two ways: they either buy large chassis switches for the spines or they break out the cables from high-speed links into

multiple lower-speed links, and build equivalent capacity networks by using multiple spines. For example, a 32-port 40 Gbps switch can typically be broken into a 96-port 10 Gbps switch. This means that the number of servers that can be supported now becomes 40 * 48 * 96 = 184,320. A 32-port 100 Gbps switch can typically be broken out into 128 25 Gbps links, with an even higher server count: 40 * 64 * 128 = 327,680. In such a three-tier network, every ToR is connected to 64 leaves, with each leaf being connected to 64 spines.

This is fundamentally the beauty of a Clos network: like fractal design, larger and larger pieces are assembled from essentially the same building blocks. Web-scale companies don't hesitate to go to 4-tier or even 6-tier Clos networks to work around the scale limitations of smaller building blocks. Coupled with the ever-larger port count support coming in merchant silicon, support for even larger data centers is quite feasible.

Crucial Side Effects of Clos Networks

Rather than relying on seemingly infallible network switches, the web-scale pioneers built resilience into their applications, thus making the network do what it does best: provide good connectivity through a rich, high-capacity connectivity matrix. As we discussed earlier, this high capacity and dense interconnect reduces the blast radius of a failure.

A consequence of using fixed-form factor switches is that there are a lot of cables to manage. The larger network operators all have some homegrown cable verification technology. There is an open source project called Prescriptive Topology Manager (PTM) that I coauthored, which handles cable verification.

Another consequence of fixed-form switches is that they fail in simple ways. A large chassis can fail in complex ways because there are so many "moving parts." Simple failures make for simpler troubleshooting, and, better still, for affordable sparing, allowing operators to swap-out failing switches with good ones instead of troubleshooting a failure in a live network. This further adds to the resilience of the network.

In other words, resilience becomes an emergent property of the parts working together rather than a feature of each box.

Building a large network with only fixed-form switches also means that inventory management becomes simple. Because any network switch is like any other, or there are at most a couple of variations, it is easy to stock spare devices and replace a failed one with a working one. This makes the network switch or router inventory model similar to the server inventory model.

These observations are important because they affect the day-to-day life of a network operator. Often, we don't integrate a new environment or choice into all aspects of our thinking. These second-order derivatives of the Clos network help a network operator to reconsider the day-to-day management of networks differently than they did previously.

Network Architecture of Clos Networks

A Clos network also calls for a different network architecture from traditional deployments. This understanding is fundamental to everything that follows because it helps understand the ways in which network operations need to be different in a data center network, even though the networking protocols remain the same.

In a traditional network, what we call leaf–spine layers were called *access-aggregation layers* of the network. These first two layers of network were connected using bridging rather than routing. Bridging uses the Spanning Tree Protocol (STP), which breaks the rich connectivity matrix of a Clos network into a loop-free tree. For example, in Figure 1-1, the two-tier Clos network, even though there are four paths between the leftmost leaf and the rightmost leaf, STP can utilize only one of the paths. Thus, the topology reduces to something like the one shown in Figure 1-3.

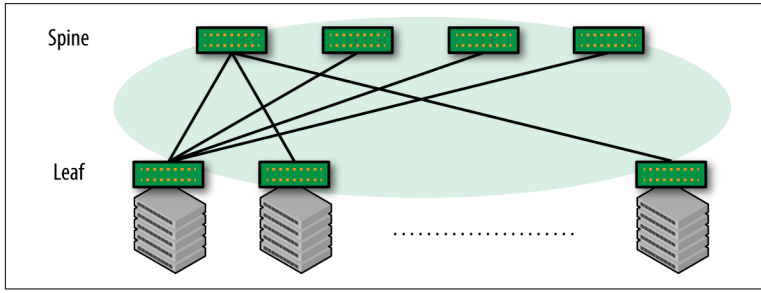


Figure 1-3. Connectivity with STP

In the presence of link failures, the path traversal becomes even more inefficient. For example, if the link between the leftmost leaf and the leftmost spine fails, the topology can look like Figure 1-4.

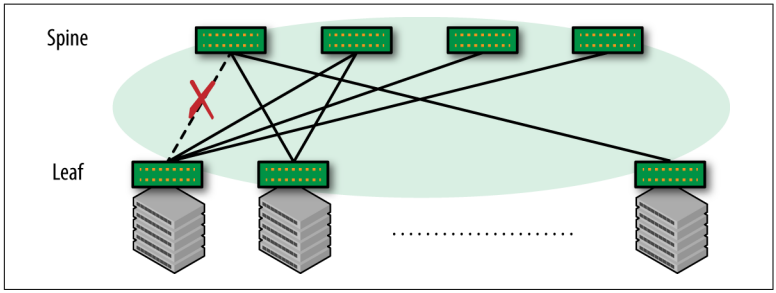


Figure 1-4. STP after a link failure

Draw the path between a server connected to the leftmost leaf and a server connected to the rightmost leaf. It zigzags back and forth between racks. This is highly inefficient and nonuniform connectivity.

Routing, on the other hand, is able to utilize all paths, taking full advantage of the rich connectivity matrix of a Clos network. Routing also can take the shortest path or be programmed to take a longer path for better overall link utilization.

Thus, the first conclusion is that routing is best suited for Clos networks, and bridging is not.

A key benefit gained from this conversion from bridging to routing is that we can shed the multiple protocols, many proprietary, that are required in a bridged network. A traditional bridged network is typically running STP, a unidirectional link detection protocol (though this is now integrated into STP), a virtual local-area network (VLAN) distribution protocol, a first-hop routing protocol such as Host Standby Routing Protocol (HSRP) or Virtual Router Redundancy Protocol (VRRP), a routing protocol to connect multiple bridged networks, and a separate unidirectional link detection protocol for the routed links. With routing, the only control plane protocols we have are a routing protocol and a unidirectional link detection protocol. That's it. Servers communicating with the first-hop router will have a simple anycast gateway, with no other additional protocol necessary.

By reducing the number of protocols involved in running a network, we also improve the network's resilience. There are fewer moving parts and therefore fewer points to troubleshoot. It should now be clear how Clos networks enable the building of not only highly scalable networks, but also very resilient networks.

Server Attach Models

Web-scale companies deploy *single-attach servers*—that is, each server is connected to a single leaf or ToR. Because these companies have a plenitude of servers, the loss of an entire rack due to a network failure is inconsequential. However, many smaller networks, including some larger enterprises, cannot afford to lose an entire rack of servers due to the loss of a single leaf or ToR. Therefore, they *dual-attach* servers; each link is attached to a different ToR. To simplify cabling and increase rack mobility, these two ToRs both reside in the same rack.

When servers are thus dual-attached, the dual links are aggregated into a single logical link (called port channel in networking jargon or bonds in server jargon) using a vendor-proprietary protocol. Different vendors have different names for it. Cisco calls it Virtual Port Channel (vPC), Cumulus calls it CLAG, and Arista calls it Multi-Chassis Link Aggregation Protocol (MLAG). Essentially, the server thinks it is connected to a single switch with a bond (or port channel). The two switches connected to it provide the illusion, from a protocol perspective mostly, that they're a single switch. This illusion is required to allow the host to use the standard Link Aggregation Control Protocol (LACP) protocol to create the bond. LACP assumes that the link aggregation happens for links between two nodes, whereas for increased reliability, the dual-attach servers work across three nodes: the server and the two switches to which it is connected. Because every multinode LACP protocol is vendor proprietary, hosts do not need to be modified to support multinode LACP. Figure 1-5 shows a dual-attached server with MLAG.

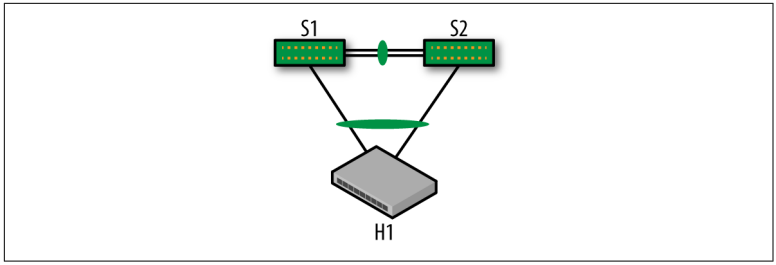


Figure 1-5. Dual-attach with port channel

Connectivity to the External World

How does a data center connect to the outside world? The answer to this question ends up surprising a lot of people. In medium to large networks, this connectivity happens through what are called *border ToRs* or *border pods*. Figure 1-6 presents an overview.

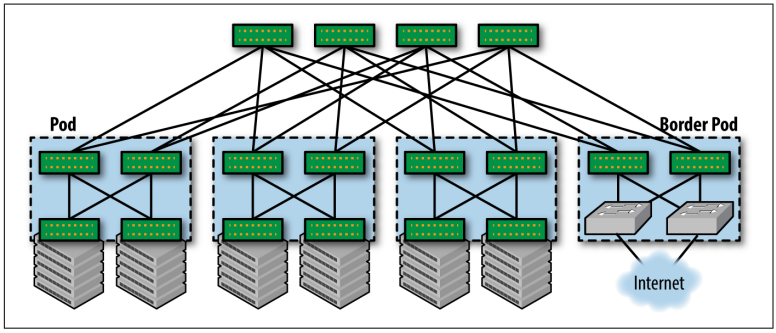


Figure 1-6. Connecting a Clos network to the external world via a border pod

The main advantage of border pods or border leaves is that they isolate the inside of the data center from the outside. The routing protocols that are inside the data center never interact with the external world, providing a measure of stability and security.

However, smaller networks might not be able to dedicate separate switches just to connect to the external world. Such networks might connect to the outside world via the spines, as shown in Figure 1-7. The important point to note is that *all* spines are connected to the internet, not some. This is important because in a Clos topology, all spines are created equal. If the connectivity to the external world were via only some of the spines, those spines would become

congested due to excess traffic flowing only through them and not the other spines. Furthermore, this would make the resilience more fragile given that losing even a fraction of the links connecting to these special spines means that either those leaves will lose complete access to the external world or will be functioning suboptimally because their bandwidth to the external world will be reduced significantly by the link failures.

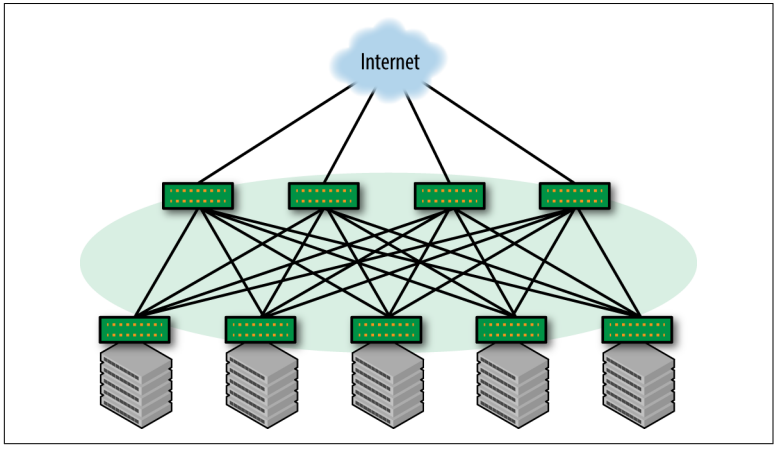


Figure 1-7. Connecting a Clos network to the external world via spines

Support for Multitenancy (or Cloud)

The Clos topology is also suited for building a network to support clouds, public or private. The additional goals of a cloud architecture are as follows:

Agility
: Given the typical use of the cloud, whereby customers spin up and tear down networks rapidly, it is critical that the network be able to support this model.

Isolation
: One customer's traffic must not be seen by another customer.

Scale
: Large numbers of customers, or tenants, must be supported.

Traditional solutions dealt with multitenancy by providing the isolation in the network, via technologies such as VLANs. Service providers also solved this problem using virtual private networks

(VPNs). However, the advent of server virtualization, aka VMs, and now containers, have changed the game. When servers were always physical, or VPNs were not provisioned within seconds or minutes in service provider networks, the existing technologies made sense. But VMs spin up and down faster than any physical server could, and, more important, this happens without the switch connected to the server ever knowing about the change. If switches cannot detect the spin-up and spin-down of VMs, and thereby a tenant network, it makes no sense for the switches to be involved in the establishment and tear-down of customer networks.

With the advent of Virtual eXtensible Local Area Network (VXLAN) and IP-in-IP tunnels, cloud operators freed the network from having to know about these virtual networks. By tunneling the customer packets in a VXLAN or IP-in-IP tunnel, the physical network continued to route packets on the tunnel header, oblivious to the inner packet's contents. Thus, the Clos network can be the backbone on which even cloud networks are built.

Operational Consequences of Modern Data Center Design

The choices made in the design of modern data centers have far reaching consequences on data center administration.

The most obvious one is that given the sheer scale of the network, it is not possible to manually manage the data centers. Automation is nothing less than a requirement for basic survival. Automation is much more difficult, if not impractical, if each building block is handcrafted and unique. Design patterns must be created so that automation becomes simple and repeatable. Furthermore, given the scale, handcrafting each block makes troubleshooting problematic.

Multitenant networks such as clouds also need to spin up and tear down virtual networks quickly. Traditional network designs based on technologies such as VLAN neither scale to support a large number of tenants nor can be spun up and spun down quickly. Furthermore, such rapid deployment mandates automation, potentially across multiple nodes.

Not only multitenant networks, but larger data centers also require the ability to roll out new racks and replace failed nodes in timescales an order or two of magnitude smaller than is possible with

traditional networks. Thus, operators need to come up with solutions that enable all of this.

Choice of Routing Protocol

It seems obvious that Open Shortest Path First (OSPF) or Intermediate System–to–Intermediate System (IS-IS) would be the ideal choice for a routing protocol to power the data center. They're both designed for use within an enterprise, and most enterprise network operators are familiar with managing these protocols, at least OSPF. OSPF, however, was rejected by most web-scale operators because of its lack of multiprotocol support. In other words, OSPF required two separate protocols, similar mostly in name and basic function, to support both IPv4 and IPv6 networks.

In contrast, IS-IS is a far better regarded protocol that can route both IPv4 and IPv6 stacks. However, good IS-IS implementations are few, limiting the administrator's choices. Furthermore, many operators felt that a link-state protocol was inherently unsuited for a richly connected network such as the Clos topology. Link-state protocols propagated link-state changes to even far-flung routers—routers whose path state didn't change as a result of the changes.

BGP stepped into such a situation and promised something that the other two couldn't offer. BGP is mature, powers the internet, and is fundamentally simple to understand (despite its reputation to the contrary). Many mature and robust implementations of BGP exist, including in the world of open source. It is less chatty than its link-state cousins, and supports multiprotocols (i.e., it supports advertising IPv4, IPv6, Multiprotocol Label Switching (MPLS), and VPNs natively). With some tweaks, we can make BGP work effectively in a data center. Microsoft's Azure team originally led the charge to adapt BGP to the data center. Today, most customers I engage with deploy BGP.

The next part of our journey is to understand how BGP's traditional deployment model has been modified for use in the data center.

CHAPTER 2

How BGP Has Been Adapted to the Data Center

Before its use in the data center, BGP was primarily, if not exclusively, used in service provider networks. As a consequence of its primary use, operators cannot use BGP inside the data center in the same way they would use it in the service provider world. If you're a network operator, understanding these differences and their reason is important in preventing misconfiguration.

The dense connectivity of the data center network is a vastly different space from the relatively sparse connectivity between administrative domains. Thus, a different set of trade-offs are relevant inside the data center than between data centers. In the service provider network, stability is preferred over rapid notification of changes. So, BGP typically holds off sending notifications about changes for a while. In the data center network, operators want routing updates to be as fast as possible. Another example is that because of BGP's default design, behavior, and its nature as a path-vector protocol, a single link failure can result in an inordinately large number of BGP messages passing between all the nodes, which is best avoided. A third example is the default behavior of BGP to construct a single best path when a prefix is learned from many different Autonomous System Numbers (ASNs), because an ASN typically represents a separate administrative domain. But inside the data center, we want multiple paths to be selected.

Two individuals put together a way to fit BGP into the data center. Their work is documented in RFC 7938.

This chapter explains each of the modifications to BGP's behavior and the rationale for the change. It is not uncommon to see network operators misconfigure BGP in the data center to deleterious effect because they failed to understand the motivations behind BGP's tweaks for the data center.

How Many Routing Protocols?

The simplest difference to begin with is the number of protocols that run within the data center. In the traditional model of deployment, BGP learns of the prefixes to advertise from another routing protocol, usually Open Shortest Path First (OSPF), Intermediate System–to–Intermediate System (IS-IS), or Enhanced Interior Gateway Routing Protocol (EIGRP). These are called internal routing protocols because they are used to control routing within an enterprise. So, it is not surprising that people assume that BGP needs another routing protocol in the data center. However, in the data center, BGP *is the* internal routing protocol. There is no additional routing protocol.

Internal BGP or External BGP

One of the first questions people ask about BGP in the data center is which BGP to use: internal BGP (iBGP) or external BGP (eBGP). Given that the entire network is under the aegis of a single administrative domain, iBGP seems like the obvious answer. However, this is not so.

In the data center, eBGP is the most common deployment model. The primary reason is that eBGP is simpler to understand and deploy than iBGP. iBGP can be confusing in its best path selection algorithm, the rules by which routes are forwarded or not, and which prefix attributes are acted upon or not. There are also limitations in iBGP's multipath support under certain conditions: specifically, when a route is advertised by two different nodes. Overcoming this limitation is possible, but cumbersome.

A newbie is also far more likely to be confused by iBGP than eBGP because of the number of configuration knobs that need to be

twiddled to achieve the desired behavior. Many of the knobs are incomprehensible to newcomers and only add to their unease.

A strong nontechnical reason for choosing eBGP is that there are more full-featured, robust implementations of eBGP than iBGP. The presence of multiple implementations means a customer can avoid vendor lock-in by choosing eBGP over iBGP. This was especially true until mid-2012 or so, when iBGP implementations were buggy and less full-featured than was required to operate within the data center.

ASN Numbering

Autonomous System Number (ASN) is a fundamental concept in BGP. Every BGP speaker must have an ASN. ASNs are used to identify routing loops, determine the best path to a prefix, and associate routing policies with networks. On the internet, each ASN is allowed to speak authoritatively about particular IP prefixes. ASNs come in two flavors: a two-byte version and a more modern four-byte version.

The ASN numbering model is different from how they're assigned in traditional, non-data center deployments. This section covers the concepts behind how ASNs are assigned to routers within the data center.

If you choose to follow the recommended best practice of using eBGP as your protocol, the most obvious ASN numbering scheme is that every router is assigned its own ASN. This approach leads to problems, which we'll talk about next. However, let's first consider the numbers used for the ASN. In internet peering, ASNs are publicly assigned and have well-known numbers. But most routers within the data center will rarely if ever peer with a router in a different administrative domain (except for the border leaves described in Chapter 1). Therefore, ASNs used within the data center come from the private ASN number space.

Private ASNs

A private ASN is one that is for use outside of the global internet. Much like the private IP address range of 10.0.0.0/8, private ASNs are used in communication between networks not exposed to the external world. A data center is an example of such a network.

Nothing stops an operator from using the public ASNs, but this is not recommended for two major reasons.

The first is that using global ASNs might confuse operators and tools that attempt to decode the ASNs into meaningful names. Because many ASNs are well known to operators, an operator might very well become confused, for example, on seeing Verizon's ASN on a node within the data center.

The second reason is to avoid the consequences of accidentally leaking out the internal BGP information to an external network. This can wreak havoc on the internet. For example, if a data center used Twitter's ASN internally, and accidentally leaked out a route claiming, say, that Twitter was part of the AS_PATH[1] for a publicly reachable route within the data center, the network operator would be responsible for a massive global hijacking of a well-known service. Misconfigurations are the number one or number two source of all network outages, and so avoiding this by not using public ASNs is a good thing.

The old-style 2-byte ASNs have space for only about 1,023 private ASNs (64512–65534). What happens when a data center network has more than 1,023 routers? One approach is to unroll the BGP knob toolkit and look for something called *allowas-in*. Another approach, and a far simpler one, is to switch to 4-byte ASNs. These new-fangled ASNs come with support for almost 95 million private ASNs (4200000000–4294967294), more than enough to satisfy a data center of any size in operation today. Just about every routing suite, traditional or new, proprietary or open source, supports 4-byte ASNs.

The Problems of Path Hunting

Returning to how the ASNs are assigned to a BGP speaker, the most obvious choice would be to assign a separate ASN for every node. But this approach leads to problems inherent to path-vector protocols. Path-vector protocols suffer from a variation of a problem called *count-to-infinity*, suffered by distance vector protocols. Although we cannot get into all the details of path hunting here, you

1 This is additional information passed with every route, indicating the list of ASNs traversed from the origin of this advertisement.

can take a look at a simple explanation of the problem from the simple topology shown in Figure 2-1.

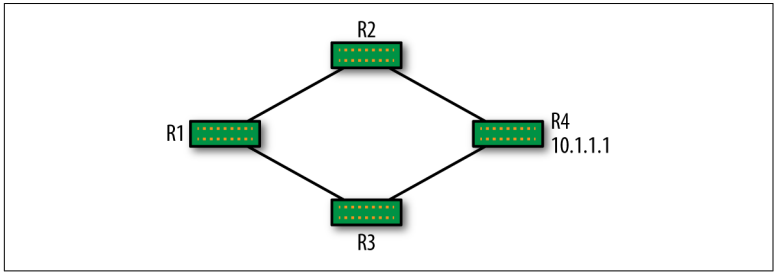


Figure 2-1. A sample topology to explain path hunting

In this topology, all of the nodes have separate ASNs. Now, consider the reachability to prefix 10.1.1.1 from R1's perspective. R2 and R3 advertise reachability to the prefix 10.1.1.1 to R1. The AS_PATH advertised by R2 for 10.1.1.1 is [R2, R4], and the AS_PATH advertised by R3 is [R3, R4]. R1 does not know how R2 and R3 themselves learned this information. When R1 learns of the path to 10.1.1.1 from both R2 and R3, it picks one of them as the best path. Due to its local support for multipathing, its forwarding tables will contain reachability to 10.1.1.1 via both R2 and R3, but in BGP's best path selection, only one of R2 or R3 can win.

Let's assume that R3 is picked as the best path to 10.1.1.1 by R1. R1 now advertises that it can reach 10.1.1.1 with the AS_PATH [R1, R3, R4] to R2. R2 accepts the advertisement, but does not consider it a better path to reach 10.1.1.1, because its best path is the shorter AS_PATH R4.

Now, when the node R4 dies, R2 loses its best path to 10.1.1.1, and so it recomputes its best path via R1, AS_PATH [R1, R3, R4] and sends this message to R1. R2 also sends a route withdrawal message for 10.1.1.1 to R1. When R3's withdrawal to route 10.1.1.1 reaches R1, R1 also withdraws its route to 10.1.1.1 and sends its withdrawal to R2. The exact sequence of events might not be as described here due to the timing of packet exchanges between the nodes and how BGP works, but it is a close approximation.

The short version of this problem is this: because a node does not know the physical link state of every other node in the network, it doesn't know whether the route is truly gone (because the node at the end went down itself) or is reachable via some other path. And

so, a node proceeds to hunt down reachability to the destination via all its other available paths. This is called path hunting.

In the simple topology of Figure 2-1, this didn't look so bad. But in a Clos topology, with its dense interconnections, this simple problem becomes quite a significant one with a lot of additional message exchanges and increased loss of traffic loss due to misinformation propagating for a longer time than necessary.

ASN Numbering Model

To avoid the problem of path hunting, the ASN numbering model for routers in a Clos topology is as follows:

- All ToR routers are assigned their own ASN.
- Leaves across a pod have a different ASN, but leaves within each pod have an ASN that is unique to that pod.
- Interpod spines share a common ASN.

Figure 2-2 presents an example of ASN numbering for a three-tier Clos.

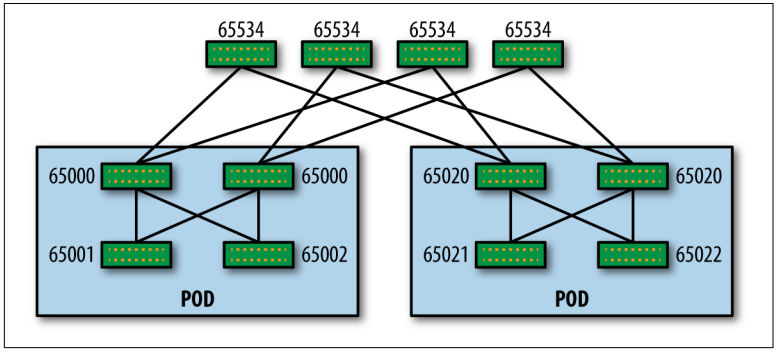


Figure 2-2. Sample ASN numbering in a Clos topology

This numbering solves the path hunting problem. In BGP, ASN is how one neighbor knows another. In Figure 2-1, let R2 and R3 be given the same ASN. When R1 told R2 that it had a path to 10.1.1.1 via R3, R2 rejected that path completely because the AS_PATH field contained the ASN of R3, which was the same as R2, which indicated a routing loop. Thus, when R2 and R3 lose their link to R4, and hence to 10.1.1.1, the only message exchange that happens is that they withdraw their advertisement to 10.1.1.1 from R1 and 10.1.1.1

is purged from all the routers' forwarding tables. In contrast, given the numbering in Figure 2-2, leaves and spines will eliminate alternate paths due to the AS_PATH loop-detection logic encoded in BGP's best-path computation.

The one drawback of this form of ASN numbering is that route aggregation or summarization is not possible. To understand why, let's go back to Figure 2-1, with R2 and R3 having the same ASN. Let's further assume that R2 and R3 have learned of other prefixes, say from 10.1.1.2/32-10.1.1.250/32 via directly attached servers (not shown in the figure). Instead of announcing 250 prefixes (10.1.1.1–10.1.1.250) to R1, both R2 and R3 decide to aggregate the routes and announce a single 10.1.1.0/24 route to R4. Now, if the link between R2 and R4 breaks, R2 no longer has a path to 10.1.1.1/32. It cannot use the path R1-R3-R4 to reach 10.1.1.1, as explained earlier. R1 has computed two paths to reach 10.1.1.0/24, via R2 and R3. If it receives a packet destined to 10.1.1.1, it might very well choose to send it to R2, which has no path to reach 10.1.1.1; the packet will be dropped by R2, causing random loss of connectivity to 10.1.1.1. If instead of summarizing the routes, R2 and R3 sent the entire list of 250 prefixes separately, when the link to R4 breaks, R2 needs to withdraw only the route to 10.1.1.1, while retaining the advertisement to the other 249 routes. R1 will correctly establish a single reachability to 10.1.1.1, via R3; but it maintains multiple paths, via R2 and R3, for the other 249 prefixes. Thus, route summarization is not possible with this ASN numbering scheme.

Best Path Algorithm

BGP uses an algorithm to compute the best path to a given prefix from a node. Understanding this is fundamental to understanding how forwarding happens in a BGP routed network, and why certain paths are chosen over others.

BGP's best path selection is triggered when a new UPDATE message is received from one or more of its peers. Implementations can choose to buffer the triggering of this algorithm so that a single run will process all updates instead of swapping routes rapidly by running the algorithm very frequently.

OSPF, IS-IS, and other routing protocols have a simple metric by which to decide which of the paths to accept. BGP has eight!

Although I'll mention them all in this section, only one matters for the data center: AS_PATH.

You can use this pithy mnemonic phrase to remember the BGP path algorithms:

Wise Lip Lovers Apply Oral Medication Every Night.

I first heard this at a presentation given by my friend and noted BGP expert, Daniel Walton. The actual inventor of the phrase is a Cisco engineer, Denise Fishburne, who was kind enough to let me use it in this book. Figure 2-3 illustrates the correspondence between the mnemonic and the actual algorithms.

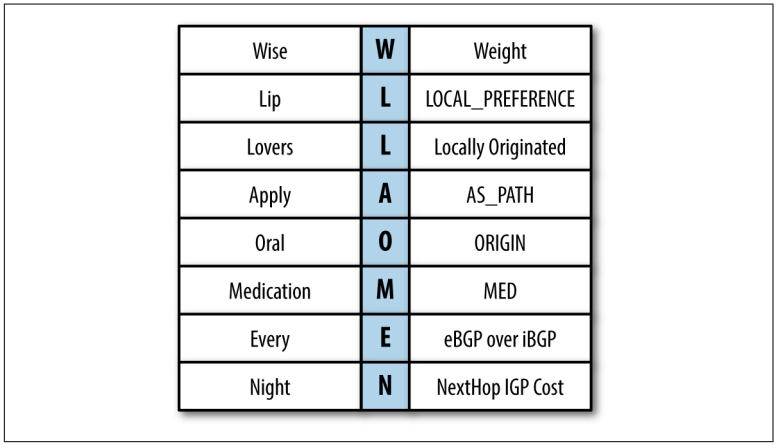


Figure 2-3. BGP best-path selection criteria

For those interested in knowing more, Section 9 of RFC 4271 covers each metric in gory detail. iBGP routes have a further match criteria beyond these eight parameters, but a discussion of those parameters is beyond the scope of this book.

Multipath Selection

In a densely connected network such as a Clos network, route multipathing is a fundamental requirement to building robust, scalable networks. BGP supports multipathing, whether the paths have equal costs or unequal costs, though not all implementations support unequal-cost multipathing. As described in the previous section, two paths are considered equal if they are equal in each of the eight criteria. One of the criteria is that the AS numbers in the AS_PATH match exactly, not just that they have equal-length paths. This

breaks multipathing in two common deployment scenarios within the data center.

The first deployment scenario, in which the same route might be announced from different ASNs, is when a server is dual-attached, with a separate ASN for each ToR switch, as shown in Figure 2-4. In the figure, the ellipses represent a bond or port channel; that is, the two links are made to look as one higher-speed logical link to upper layer protocols.

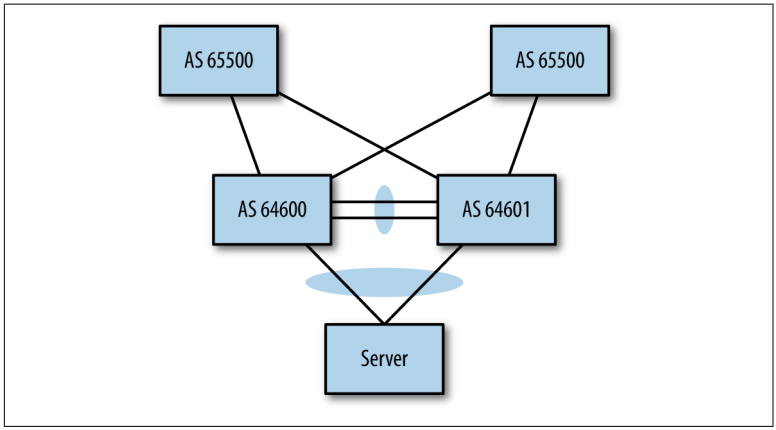


Figure 2-4. Dual-attached server

Let's assume that both leaves announce a subnet route to 10.1.1.0/24, the subnet of the bridge to which the server is attached. In this case, each spine sees the route to 10.1.1.0/24 being received, one with AS_PATH of 64600, and the other with an AS_PATH of 64601. As per the logic for equal-cost paths, BGP requires not only that the AS_PATH lengths be the same, but that the AS_PATHs contain the same ASN list. Because this is not the case here, each spine will not multipath; instead, they will pick only one of the two routes.

In the second deployment scenario, when virtual services are deployed by servers, multiple servers will announce reachability to the same service virtual IP address. Because the servers are connected to different switches to ensure reliability and scalability, the spines will again receive a route from multiple different ASNs, for which the AS_PATH lengths are identical, but the specific ASNs inside the path itself are not.

There are multiple ways to address this problem, but the simplest one is to configure a knob that modifies the best-path algorithm. The knob is called `bestpath as-path multipath-relax`. What it does is simple: when the AS_PATH lengths are the same in advertisements from two different sources, the best-path algorithm skips checking for exact match of the ASNs, and proceeds to match on the next criteria.

Slow Convergence Due to Default Timers

To avoid configuring every knob explicitly, a common practice is to assume safe, conservative values for parameters that are not specified. Timers in particular are a common knob for which defaults are assumed if the operator doesn't provide any specific information. In the simplest terms, timers control the speed of communication between the peers. For BGP, these timers are by default tuned for the service provider environment, for which stability is preferred over fast convergence. Inside the data center, although stability is certainly valued, fast convergence is even more important.

There are four timers that typically govern how fast BGP converges when either a failure occurs or when it is recovering from a failure (such as a link becoming available again). Understanding these timers is important because they affect the speed with which the information propagates through the network, and tuning them allows an operator to achieve convergence speeds with BGP that match other internal routing protocols such as Open Shortest Path First (OSPF). We'll look at these timers in the following sections.

Advertisement Interval

BGP maintains a minimum interval per neighbor. Events within this minimum interval window are bunched together and sent at one shot when the minimum interval expires. This is essential for the most stable code, but it also helps prevent unnecessary processing in the event of multiple updates within a short duration. The default value for this interval is 30 seconds for eBGP peers, and 0 seconds for iBGP peers. However, waiting 30 seconds between updates is entirely the wrong choice for a richly connected network such as those found in the data center. 0 is the more appropriate choice because we're not dealing with routers across administrative

domains. This change alone can bring eBGP's convergence time to that of other IGP protocols such as OSPF.

Keepalive and Hold Timers

In every BGP session, a node sends periodic keepalive messages to its peer. If the peer doesn't receive a keepalive for a period known as the hold time, the peer declares the session as dead, drops the connection and all the information received on this connection, and attempts to restart the BGP state machine.

By default, the keepalive timer is 60 seconds and the hold timer is 180 seconds. This means that a node sends a keepalive message for a session every minute. If the peer does not see a single keepalive message for three minutes, it declares the session dead. By default, for eBGP sessions for which the peer is a single routing hop away, if the link fails, this is detected and the session is reset immediately. What the keepalive and hold timers do is to catch any software errors whereby the link is up but has become one-way due to an error, such as in cabling. Some operators enable a protocol called Bidirectional Forwarding Detection (BFD) for subsecond, or at most a second, detection of errors due to cable issues. However, to catch errors in the BGP process itself, you need to adjust these timers.

Inside the data center, three minutes is a lifetime. The most common values configured inside the data center are three seconds for keepalive and nine seconds for the hold timer.

Connect Timer

This is the least critical of the four timers. When BGP attempts to connect with a peer but fails due to various reasons, it waits for a certain period of time before attempting to connect again. This period by default is 60 seconds. In other words, if BGP is unable to establish a session with its peer, it waits for a minute before attempting to establish a session again. This can delay session reestablishment when a link recovers from a failure or a node powers up.

Default Configuration for the Data Center

When crossing administrative and trust boundaries, it is best to explicitly configure all of the relevant information. Furthermore, given the different expectations of two separate enterprises, almost

nothing is assumed in BGP, with every knob needing to be explicitly configured.

When BGP was adapted for use in the data center, none of these aspects of BGP was modified. It is not the protocol itself that needs to be modified, but the way it is configured. Every knob that must be configured strikes terror (or at least potentially sows confusion) in the minds of newbies and intermediate practitioners. Even those who are versed in BGP feel the need to constantly keep up because of the amount of work required by BGP.

A good way to avoid all of these issues is to set up good defaults so that users don't need to know about the knobs they don't care about. The BGP implementation in many proprietary routing suites originated in the service provider world, so such an option is not typically available. With open source routing suites that are geared toward the data center, such as FRRouting, the default configuration saves the user from having to explicitly configure many options.

Good defaults also render the size of your configuration much more manageable, making it easy to eyeball configurations and ensure that there are no errors. As your organization becomes more familiar with BGP in the data center, sane default configurations can provide the basis for reliable automation.

Here are the default settings in FRRouting for BGP. These are the settings I believe are the best practice for BGP in the data center. These are the settings I've seen used in just about every production data center I've encountered.

- Multipath enabled for eBGP and iBGP
- Advertisement interval set to 0
- Keepalive and Hold timers set to 3s and 9s
- Logging adjacency changes enabled

Summary

This chapter covered the basic concepts behind adapting BGP to the data center, such as the use of eBGP as the default deployment model and the logic behind configuring ASNs. In the next two chapters, we'll apply what we learned in this chapter to configuring nodes in a Clos topology.

CHAPTER 3

Building an Automatable BGP Configuration

It is not sufficient to merely learn how to configure BGP. A network operator also needs to know how to go about automating the deployment of this configuration.

As discussed in "Operational Consequences of Modern Data Center Design" on page 13, the mantra of automation in the data center is simple: automate or die. If you cannot automate your infrastructure—and the network is a fundamental part of the infrastructure—you'll simply become too inefficient to meet the business objectives. As a consequence, either the business will shrivel up or evolve to improve its infrastructure.

In this chapter, we begin the journey of building an automatable BGP configuration. We won't show automation with any particular tool such as Ansible, because sites vary in their use of these tools and each has its own syntax and semantics that deserve their own documentation. Instead, we'll focus on BGP.

The Basics of Automating Configuration

Automation is possible when there are patterns. If we cannot find patterns, automation becomes extremely difficult, if not impossible. Configuring BGP is no different. We must seek patterns in the BGP configuration so that we can automate them. However, detecting patterns isn't sufficient. The patterns need to be robust so that

changes don't become hazardous. We must also avoid duplication. In the section that follows, we'll examine both of these problems in detail, and see how we can eliminate them.

Sample Data Center Network

For much of the rest of the book, we'll use the topology in Figure 3-1 to show how to use BGP. This topology is a good representation of most data center networks.

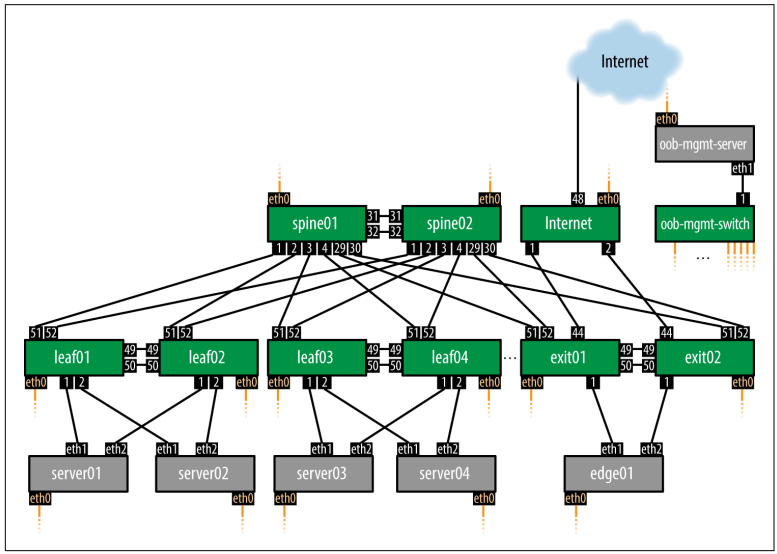


Figure 3-1. Sample data center network

In our network, we configure the following:

- The leaves, leaf01 through leaf04
- The spines, spine01 through spine02
- The exit leaves, exit01 through exit02
- The servers, server01 through server04

Except for the servers, all of the devices listed are routers, and the routing protocol used is BGP.

NOTE

A quick reminder: the topology we are using is a Clos network, so the leaf and spine nodes are all routers, as described in Chapter 1.

Interface Names Used in This Book

Interface names are specific to each routing platform. Arista, Cisco, Cumulus, and Juniper all have their own ways to name an interface. In this book, I use the interface names used on Cumulus Linux. These ports are named swpX, where swp stands for switchport. So, in Figure 3-1, server01's eth1 interface is connected to leaf01's swp1 interface. Similarly, leaf01's swp51 interface is connected to spine01's swp1 interface.

This chapter configures two routers: leaf01 and spine01. We then can take this configuration and apply it to other spine and leaf nodes with their specific IP addresses and BGP parameters.

The Difficulties in Automating Traditional BGP

Example 3-1 shows the simplest possible configurations of leaf01 and leaf02. For those who are new to BGP, a few quick words about some of the key statements in the configuration:

`router bgp 65000`
: This is how you specify the ASN for this BGP speaker. This also marks the start of the BGP-specific configuration block in FRR.

`bgp router-id 10.0.254.1`
: Every routing protocol speaker has a unique `router-id` that identifies the speaker. This is true across all routing protocols, including BGP. Bad things ensue if this ID is not unique in most protocols, so it's just good practice to keep this unique by making it the same as the loopback IP address.

`neighbor peer-group ISL`
: In FRR, this is a way to define a configuration template.

`neighbor ISL remote-as 65500`
: This is the specification of the remote end's ASN. Traditional BGP configurations require this. We'll see how we can simplify this in the next chapter.

`neighbor 169.254.1.0 peer-group ISL`
: This is how you indicate to the BGP daemon that you'd like to establish a session with the specified IP address, using the parameters specified in the configuration template ISL.

`address-family ipv4 unicast`
: Given that BGP is a multiprotocol routing protocol, the `address-family` block specifies the configuration to apply for a specific protocol (in this case, `ipv4 unicast`).

`neighbor ISL activate`
: BGP requires you to explicitly state that you want it to advertise routing state for a given address family' and that is what `activate` does.

`network 10.0.254.1/32`
: This tells BGP to advertise reachability to the prefix 10.0.254.1/32. This prefix needs to already be in the routing table in order for BGP to advertise it.

`maximum-paths 64`
: This tells BGP that it needs to use multiple paths, if available, to reach a prefix.

The meaning of the various timers was discussed in "Slow Convergence Due to Default Timers" on page 24.

Example 3-1. Highlighting the router-specific configuration across leaf01 and leaf02

```
// leaf01's BGP configuration

log file /var/log/frr/frr.log

router bgp 65000
  bgp router-id 10.0.254.1
  bgp log-neighbor-changes
  bgp no default ipv4-unicast
  timers bgp 3 9
  neighbor peer-group ISL
  neighbor ISL remote-as 65500
```

```
  neighbor ISL advertisement-interval 0
  neighbor ISL timers connect 5
  neighbor 169.254.1.0 peer-group ISL
  neighbor 169.254.1.64 peer-group ISL
  address-family ipv4 unicast
    neighbor ISL activate
    network 10.0.254.1/32
    network 10.1.1.0/26
    maximum-paths 64
  exit-address-family

// leaf02's BGP configuration

log file /var/log/frr/frr.log

router bgp 65001
  bgp router-id 10.0.254.2
  bgp log-neighbor-changes
  bgp no default ipv4-unicast
  timers bgp 3 9
  neighbor peer-group ISL
  neighbor ISL remote-as 65500
  neighbor ISL advertisement-interval 0
  neighbor ISL timers connect 5
  neighbor 169.254.1.0 peer-group ISL
  neighbor 169.254.1.64 peer-group ISL
  address-family ipv4 unicast
    neighbor ISL activate
    network 10.0.254.1/32
    network 10.1.1.0/26
    maximum-paths 64
  exit-address-family
```

Let's look at leaf01 by itself first to see what is duplicated in it. For example, 10.0.254.1 is specified twice, once with /32 and once without. The first time it is specified as the default gateway address, and the second time as the interface.

Configuration is less error-prone when there is as little duplication as possible. It is a well-known maxim in coding to avoid duplicating code. Duplication is problematic because with more places to fix the same piece of information, it is easy to forget to fix one of the multiple places when making a change or fixing a problem. Duplication is also cumbersome because a single change translates to changes needing to be made in multiple places.

Consider the effects of duplicating the IP address across the interface and inside BGP. If the interface IP address changes, a corresponding change must be made in the BGP configuration, as well.

Otherwise, you'll lose connectivity after the change. Another example is you we changed the default gateway address on this node and assigned it to another node, but forgot the change the `router-id`. You'd end up with two routers having the same `router-id`, which could result in peering difficulties (though only in iBGP, not eBGP). The same thing would apply for the network statements, too.

Furthermore, this configuration assumes just a single VLAN or subnet for each of the leaves. If there were multiple subnets, individually listing them all would be unscalable. Or, even if you did that, the resulting configuration would be too long to be readable.

Now let's compare the configuration across the spines, as shown in Example 3-2.

Example 3-2. Highlighting the router-specific configuration across spine01 and spine02

```
// spine01's BGP configuration

log file /var/log/frr/frr.log

router bgp 65534
  bgp router-id 10.0.254.254
  bgp log-neighbor-changes
  bgp no default ipv4-unicast
  timers bgp 3 9
  neighbor peer-group ISL
  neighbor ISL advertisement-interval 0
  neighbor ISL timers connect 5
  neighbor 169.254.1.1 remote-as 65000
  neighbor 169.254.1.1 peer-group ISL
  neighbor 169.254.1.3 remote-as 65001
  neighbor 169.254.1.3 peer-group ISL
  neighbor 169.254.1.5 remote-as 65002
  neighbor 169.254.1.5 peer-group ISL
  neighbor 169.254.1.5 remote-as 65003
  neighbor 169.254.1.7 peer-group ISL
  bgp bestpath as-path multipath-relax
  address-family ipv4 unicast
    neighbor ISL activate
    network 10.0.254.254/32
    maximum-paths 64
  exit-address-family

// spine02's BGP configuration

log file /var/log/frr/frr.log
```

```
router bgp 65534
  bgp router-id 10.0.254.253
  bgp log-neighbor-changes
  bgp no default ipv4-unicast
  timers bgp 3 9
  neighbor peer-group ISL
  neighbor ISL advertisement-interval 0
  neighbor ISL timers connect 5
  neighbor 169.254.1.1 remote-as 65000
  neighbor 169.254.1.1 peer-group ISL
  neighbor 169.254.1.3 remote-as 65001
  neighbor 169.254.1.3 peer-group ISL
  neighbor 169.254.1.5 remote-as 65002
  neighbor 169.254.1.5 peer-group ISL
  neighbor 169.254.1.5 remote-as 65003
  neighbor 169.254.1.7 peer-group ISL
  bgp bestpath as-path multipath-relax
  address-family ipv4 unicast
    neighbor ISL activate
    network 10.0.254.254/32
    maximum-paths 64
  exit-address-family
```

The same issues that were present in the configuration across the leaves is also present in the configuration across the spines.

However, there are a few things done right in this configuration:

- The interface IP addresses have a pattern. Assuming 32 port spines (32 × 100 Gbps and 32 × 40 Gbps switches are common these days), 64 interface IP addresses are required. Using /31 subnets across each interface allows us to allocate a /26 subnet across the two spines.
- The default gateway address subnets are announced from a common subnet, which is different from the subnets allocated to end hosts.
- Assuming 40 hosts per rack, which is itself a stretch in all but the largest data centers, in the network configuration, we allocated /26 subnets for each subnet associated to hosts.

To summarize the difficulties with the configurations across nodes, we see that using IP addresses means we duplicate information in multiple places, and so the configuration becomes fragile and unscalable as new IP addresses are added and removed.

Although detecting patterns in the neighbor IP addresses is possible, it is also fragile in that later changes can break the assumptions built in to the pattern recognition. For example, if we assume that we numbered the addresses serially, adding a new spine later can break that pattern. So, instead of the addition being simple, every change would be fragile and need to be handled specially.

How, then, do we overcome these issues? Time to unpack a few tools from the armory.

Redistribute Routes

To eliminate the specification of individual IP addresses to announce via `network` statements, we can use a different command: `redistribute`.

Since just about their first introduction, all routing protocol suites have provided an option to take prefixes from one protocol and advertise it in another. This practice is called *redistributing routes*.

The general command format in BGP looks like this:

```
redistribute protocol route-map route-map-name
```

The `protocol` is one of the following values:

`static`
: Announce routes that have been statically configured.

`connected`
: Announce routes associated with interface addresses. The links on these interfaces must be up and operational when this configuration runs. If a link fails, its IP address is withdrawn immediately.

`kernel`
: This is specific to Linux operating systems. Routes can be statically configured either by a routing suite—FRRouting, bird, or quagga, for example—or directly in the kernel, either via a toolset such as iproute2 (the `ip` family of commands) or directly via the netlink interface to the kernel itself.

`ospf`
: Redistribute routes learned via the OSPF protocol.

`bgp`
: Redistribute routes learned via the BGP protocol.

`rip`
: Redistribute routes learned via Routing Information Protocol (RIP) protocol.

Some other less common protocols also can be represented, such as IS-IS.

So, to advertise the interface IP addresses of all the VLANs on the box and default gateway, it is sufficient to replace all the network statements with a single command:

```
redistribute connected
```

The configuration on leaf01 would look like this after replacing `network` statements with `redistribute`:

```
log file /var/log/frr/frr.log
router bgp 65000
  bgp router-id 10.0.254.1
  bgp log-neighbor-changes
  bgp no default ipv4-unicast
  timers bgp 3 9
  neighbor peer-group ISL
  neighbor ISL remote-as 65500
  neighbor ISL advertisement-interval 0
  neighbor ISL timers connect 5
  neighbor 169.254.1.0 peer-group ISL
  neighbor 169.254.1.64 peer-group ISL
  address-family ipv4 unicast
    neighbor ISL activate
    redistribute connected
    maximum-paths 64
  exit-address-family
```

However, the use of an unadorned redistribute statement leads to potentially advertising addresses that should not be, such as the interface IP addresses, or in propagating configuration errors. As an example of the latter, if an operator accidentally added an IP address of 8.8.8.8/32 on an interface, the BGP will announce reachability to that address, thereby sending all requests meant for the public, well-known, DNS server to that hapless misconfigured router.

To avoid all of these issues, just about every routing protocol supports some form of routing policy.

Routing Policy

Routing policy, at its simplest, specifies when to accept or reject route advertisements. Based on where they're used, the accept or reject could apply to routes received from a peer, routes advertised to a peer, and redistributed routes. At its most complex, routing policy can modify metrics that affect the best-path selection of a prefix, and add or remove attributes or communities from a prefix or set of prefixes. Given BGP's use primarily in connecting different administrative domains, BGP has the most sophisticated routing policy constructs.

A routing policy typically consists of a sequence of if-then-else statements, with matches and actions to be taken on a successful match.

While we've thus far avoided the use of any routing policy, we can now see the reason for using them with BGP in the data center.

For example, to avoid the problem of advertising 8.8.8.8, as described in the previous section, the pseudocode for the routing policy would look like the following (we develop this pseudocode into actual configuration syntax by the end of this section):

```
if prefix equals '8.8.8.8/32' then reject else accept
```

In a configuration in which connected routes are being redistributed, a safe policy would be to accept the routes that belong to this data center and reject any others. The configurations I've shown, contain two kinds of prefixes: 10.1.0.0/16 (assuming there are lots of host-facing subnets in the network) and the router's loopback IP address, as an example 10.0.254.1/32. We also see the interface address subnet, 169.254.0.0/16, which must not be advertised. So, a first stab at a routing policy would be the following:

```
if prefix equals 10.1.0.0/16 then accept
else if prefix equals 10.0.254.1/32 then accept
else reject
```

However, this requires us to put in a different `route-map` clause for every router because every router has a different loopback IP address. If instead we chose the subnet from which these addresses are allocated, 10.0.254.0/24, the `route-map` becomes the same across all of the routers. However, because the loopback IP address of a router is contained within this subnet and is not equal to this subnet, we cannot use `prefix equals`. Instead, we introduce a new

qualifier, `belongs`, which checks whether an IP address belongs to the specified subnet. Here's the newly rewritten pseudocode for the routing policy:

```
if prefix belongs to 10.1.0.0/16 then accept
else if prefix belongs to 10.0.254.0/24 then accept
else reject
```

But this would accept anyone accidentally announcing the subnet 10.0.254.0/26, as an example, when the allowed prefixes are the precise addresses of the router loopbacks, all of which are /32 addresses. How can we address this? By adding more qualifiers:

```
if prefix belongs to 10.1.0.0/16 then accept
else if (prefix belongs to 10.0.254.0/24 and
         address mask equals 32) then
     accept
else reject
```

The qualifier we added, `address mask equals`, allows us to match addresses more precisely by accounting for not just the address, but the address mask, as well.

Because multiple such routing policies are possible, let's give this policy a name and make it a function thus:

```
ACCEPT_DC_LOCAL(prefix)
{
    if prefix belongs to 10.1.0.0/16 then accept
    else if (10.0.254.0/24 contains prefix and
             subnet equals 32) then
    accept
    else reject
}
```

NOTE

Just about every network configuration I've seen uses all caps for `route-map` and `prefix-list` names. Although this is just a name and operators are free to choose their conventions—all caps, camelCase, or anything else—it is useful to be aware of convention.

Route-Maps

`route-maps` are a common way to implement routing policies. Cisco's IOS, NXOS, the open source protocol suite FRRouting, Arista, and others support `route-maps`. JunOS uses a different syntax with, some would argue, more intuitive keywords. The open source routing suite BIRD goes a step further and uses a simple domain-specific

programming language instead of this combination of `route-maps` and `prefix-lists`. The details of describing that are beyond the scope of this book, but if you're interested, you can find the details on BIRD's web pages.

`route-maps` have the following syntax:

```
route-map NAME (permit|deny) [sequence_number]
  match classifier
  set action
```

This assigns a name to the policy, indicates whether the matched routes will be permitted or denied, and then matches inputs against a classifier. If a `match` clause successfully matches a classifier, the `set` clause acts on the route. The optional `sequence number` orders the sequence of clauses to be executed within a `route-map`.

When we use the `permit` keyword, the `set` action is applied when the match succeeds, but when we use the `deny` keyword, the `set` action is applied when the match fails. In other words, `deny` functions as a "not" operator: if there's a match, reject the route.

`route-maps` have an implicit "deny" at the end. Thus, if no entry is matched, the result is to reject the input.

Classifiers in route-maps

`route-maps` come with a rich set of classifiers. You can use an extensive variety of traits as classifiers, and different implementations support different subsets of these classifiers (some support all and more). The list in Table 3-1 is taken from FRRouting's list.

Table 3-1. Key classifiers in the match field of a `route-map`

`as-path`	Match value from BGP's AS_PATH
`community`	Match value from a prefix's community, if any
`extcommunity`	Match value from BGP's extended community list
`interface`	Match name of next hop interface of route
`ip`, `ipv6`	Match IP information such as IP address, nexthop, or source
`local-preference`	Match LOCAL_PREFERENCE of route
`metric`	Match route's metric field
`origin`	Match route's ORIGIN attribute
`peer`	Match session peer's information

As an example of a routing policy using IP prefixes as classifiers, let's begin by looking at how two prefixes are defined:

```
ip prefix-list DC_LOCAL_SUBNET seq 5 permit 10.1.0.0/16 le 26
ip prefix-list DC_LOCAL_SUBNET seq 10 permit 10.0.254.0/24 le 32
```

These commands together define a single list called `DC_LOCAL_SUBNET` that contains two prefixes: `10.1.0.0/16` and `10.0.254.0/24`. In both cases, matching any prefix against this list checks whether the prefix either matches exactly or is contained in the prefixes provided. In this case, `10.0.254.0/24 le 32` specifically states that any match must be on a subnet that is /32. Even though it says "less than or equal to," in IPv4 there's no subnet smaller than /32, and so this functions as an exact match for /32 prefixes only.

`seq <number>` is used to identify the order of matches. For example, if you wanted to reject 10.1.1.1/32 but permit 10.1.1.0/24, the right way to order the `prefix-lists` using sequence number would be as follows:

```
ip prefix-list EXAMPLE_SEQ seq 5 deny 10.1.1.1/32
ip prefix list EXAMPLE_SEQ seq 10 permit 10.1.1.0/24
```

To allow clauses to be inserted in the middle of an existing order, a common practice is to separate sequence numbers with some gap. In the example, we used a gap of 5.

Now, we can define a `route-map` to match the two prefixes with the `DC_LOCAL_SUBNET` name. The following is the `route-map` equivalent of the if-then-else route policy pseudocode described earlier in the routing policy section, and includes the `redistribute` command that takes this policy into account:

```
ip prefix-list DC_LOCAL_SUBNET seq 5 permit 10.1.0.0/16 le 26
ip prefix-list DC_LOCAL_SUBNET seq 10 permit 10.0.254.0/24 le 32
route-map ACCEPT_DC_LOCAL permit 10
  match ip-address DC_LOCAL_SUBNET

redistribute connected route-map DC_LOCAL_SUBNET
```

Here's the pseudocode equivalent of this `route-map`:

```
DC_LOCAL_SUBNET(prefix)
{
    if (prefix belongs to 10.1.1.0/26 and prefixlen <= 26 or
    prefix belongs to 10.0.254.0/24 and prefixlen <= 32) then
    redistribute connected route
}
```

Instead of IP prefixes, we can use any of the other classifiers, as well. For example, if all we need to do was advertise the router's primary loopback IP address, the config lines are as follows:

```
route-map ADV_LO permit 10
   match interface lo

redistribute connected route-map ADV_LO
```

Note that this will not advertise the host-local 127.x.x.x address associated with the loopback interface, but only the globally reachable IP addresses.

Writing secure route-map policies

There are secure and insecure ways of writing routing policy. The fundamental principle is this: always reject anything that isn't explicitly permitted. Let's consider this by looking at an example. It's not uncommon to want to advertise IP addresses on all interfaces except those on the uplink (inter-switch) interfaces `swp51` and `swp52`, and the management interface, `eth0`. Here's one way to write the configuration:

```
route-map EXCEPT_ISL_ETH0 deny 10
   match interface swp51
route-map EXCEPT_ISL_ETH0 deny 20
   match interface swp52
route-map EXCEPT_ISL_ETH0 deny 30
   match interface eth0
route-map EXCEPT_ISL_ETH0 permit 40

redistribute connected route-map EXCEPT_ISL_ETH0
```

The final `permit` configuration allows through any interface that didn't match one of the `deny` `route-maps`.

Following is the pseudocode equivalent of this `route-map`:

```
EXCEPT_ISL_ETH0(interface)
{
    if interface is not swp51 and
       interface is not swp52 and
       interface is not eth0 then
       redistribute connected
}
```

The benefit of this approach is that it allows you to change interfaces freely and use non-contiguous IP addresses for them, without changing the `route-map` or modifying BGP configuration. The dis-

advantage is that any new interface that comes up with a valid IP address will have its IP address immediately advertised, whether the administrator intended it or not. Therefore, this is considered an insecure approach that you must never use in configuring routing policy.

The alternate can be tedious if there are lots of interfaces whose addresses need to be announced. Typical routing suite implementations do not allow the specification of multiple interfaces via a syntax such as `swp1-49` (include all interfaces from `swp1` through `swp49`). In such cases, resorting to using IP addresses that might be a smaller list might be an option if the IP addressing used on the interfaces comes from only a few subnets.

route-maps in BGP

Besides redistributed routes, you can apply `route-maps` in multiple other places during BGP processing. Here are some examples:

- Using `route-maps` to filter out what prefixes to accept in an advertisement from a neighbor:

```
neighbor 169.254.1.1 route-map NBR_RT_ACCEPT in
```

- Using `route-maps` to filter out what routes what routes to advertise to a neighbor:

```
neighbor 169.254.1.1 route-map NBR_RT_ADV out
```

- Filtering out routes considered for advertisement via a `network` statement:

```
network 10.1.1.1/24 route-map ADV_NET
```

- Advertising default routes:

```
neighbor foo default-originate route-map ONLY_NON_EXITS
```

Effect of route-maps on BGP processing

BGP is a path-vector routing protocol, and so it doesn't announce route updates until it runs the best-path algorithm. `route-maps` are applied on packet receive and on packet send. If a BGP speaker has tens or hundreds of neighbors and there are `route-maps` attached to these neighbors, running the `route-map` for each neighbor before advertising the route becomes CPU-intensive and slows down the

sending of updates. Slow update processing can result in poor convergence times, too.

Therefore, `peer-groups` often are used with `route-maps` to drastically reduce the amount of processing BGP needs to do before advertising a route to its neighbors. Instead of relying on just user-configured peer groups, implementations typically build up these groups dynamically. This is because even within a single `peer-group`, different neighbors might support different capabilities (for example, some might support MPLS, and some might not). This information can be determined only during session establishment. So, user configuration either doesn't help or places an undue burden on the user to ensure that it all neighbors in a peer group support exactly the same capabilities.

Thus, an implementation that supports the dynamic creation and teardown of peer groups puts all neighbors that have the same outgoing route policy and the same capabilities in a new, dynamically created peer group or, more precisely, dynamic update group. BGP runs the policy once for a prefix that encompasses the entire peer group. The result is then automatically applied to each member of that dynamically constructed peer group. This allows implementations to scale to supporting hundreds or even thousands of neighbors.

Using Interface Names as Neighbors

Because we're using /31 addresses for interface IP addresses, it's easy to determine the interface IP address of the peer. For example, if one end has an IP address of 169.254.1.0/31, the IP address of the end of the interface is obviously 169.254.1.1/31. Similarly, if one end has an IP address of 169.254.1.64/31, the other end has an IP address of 169.254.1.65/31. The same would be true if /30 subnets were used for interface addresses.

FRRouting uses this trick to allow users to substitute the interface name in neighbor statements instead of specifying IP addresses. This changes the neighbor configuration in leaf01 from

```
neighbor 169.254.1.0 peer-group ISL
neighbor 169.254.1.64 peer-group ISL
```

to:

```
    neighbor swp51 interface peer-group ISL
    neighbor swp52 interface peer-group ISL
```

When the BGP code in FRRouting encounters an interface name in the neighbor statement, it checks to see whether the interface has an IPv4 address that is a /30 or a /31. If so, BGP automatically identifies the remote end's IP address and initiates a BGP session to that IP address. If the IP address is not a /30 or /31 and there is an IPv4 address on the link, the code prints a warning and stops trying to initiate a connection.

Using interface names instead of IP addresses makes the configuration across the leaves and spines look quite a bit alike, as Example 3-3 shows.

Example 3-3. BGP configuration of leaves using interface names

```
// leaf01's BGP configuration

log file /var/log/frr/frr.log

ip prefix-list DC_LOCAL_SUBNET 5 permit 10.1.0.0/16 le 26
ip prefix-list DC_LOCAL_SUBNET 10 permit 10.0.254.0/24 le 32
route-map ACCEPT_DC_LOCAL permit 10
  match ip-address DC_LOCAL_SUBNET

router bgp 65000
  bgp router-id 10.0.254.1
  bgp log-neighbor-changes
  bgp no default ipv4-unicast
  timers bgp 3 9
  neighbor peer-group ISL
  neighbor ISL remote-as 65500
  neighbor ISL advertisement-interval 0
  neighbor ISL timers connect 5
  neighbor swp51 peer-group ISL
  neighbor swp52 peer-group ISL
  address-family ipv4 unicast
    neighbor ISL activate
    redistribute connected route-map DC_LOCAL
    maximum-paths 64
  exit-address-family

// leaf02's BGP configuration

log file /var/log/frr/frr.log

ip prefix-list DC_LOCAL_SUBNET 5 permit 10.1.0.0/16 le 26
ip prefix-list DC_LOCAL_SUBNET 10 permit 10.0.254.0/24 le 32
```

```
route-map ACCEPT_DC_LOCAL permit 10
  match ip-address DC_LOCAL_SUBNET

router bgp 65001
  bgp router-id 10.0.254.2
  bgp log-neighbor-changes
  bgp no default ipv4-unicast
  timers bgp 3 9
  neighbor peer-group ISL
  neighbor ISL remote-as 65500
  neighbor ISL advertisement-interval 0
  neighbor ISL timers connect 5
  neighbor swp51 peer-group ISL
  neighbor swp52 peer-group ISL
  address-family ipv4 unicast
    neighbor ISL activate
    redistribute connected route-map DC_LOCAL
    maximum-paths 64
  exit-address-family
```

The configuration across the spines also looks the same, except for changes to the `router-id` and neighbor's ASN. Here is the result:

```
    log file /var/log/frr/frr.log

    ip prefix-list ACCRT 5 permit 10.1.0.0/16 le 26
    ip prefix-list ACCRT 10 permit 10.0.254.0/24 le 32
    route-map DC_LOCAL permit 10
      match ip-address ACCRT

    router bgp 65500
      bgp router-id 10.0.254.254
      bgp log-neighbor-changes
      bgp no default ipv4-unicast
      timers bgp 3 9
      neighbor peer-group ISL
      neighbor ISL advertisement-interval 0
      neighbor ISL timers connect 5
      neighbor swp1 remote-as 65000
      neighbor swp1 peer-group ISL
      neighbor swp2 remote-as 65001
      neighbor swp2 peer-group ISL
      neighbor swp3 remote-as 65002
      neighbor swp3 peer-group ISL
      neighbor swp4 remote-as 65003
      neighbor swp4 peer-group ISL
      bgp bestpath as-path multipath-relax
      address-family ipv4 unicast
        neighbor ISL activate
        redistribute connected route-map DC_LOCAL
```

```
    maximum-paths 64
  exit-address-family
```

The same routing policy is applied across the leaf and the spine.

The use of interface names instead of IP addresses is not limited just to configuration. All the relevant show and clear commands (we talk about these in Chapter 5) also can take interface names.

Using interface names instead of IP addresses supports the other cardinal principle of good configuration: reduced duplication. With both `redistribute connected` and `neighbor` statements, the only place the IP address of the interface is specified is in the interface configuration.

If you're conversant with automation, you might wonder how replacing IP addresses with interface names makes it more automation friendly. Why can't the use of variables to deal with the differences between routers be sufficient? At the end of the day, automation involves some sort of programming logic, and so depending on the tool, the logic can be simple or complex. But simplicity in the automation code is crucial in reducing errors. Many studies show that operator-caused errors are the second most common reason for network outages. With automation, we introduce a level of abstraction as well as the ability to wreak havoc across a much larger number of routers instantaneously. For example, one approach to cabling can be that the operator uses different ports on different leaves to connect inter-switch links. Because the ports on each node are different, using variables per node to define the uplink ports can make for a bad physical network design. But these variables create a level of abstraction and thereby mask the problem if operators are not careful. With uniform cabling, the operator can eliminate the need to define variables for the uplink ports across the different nodes. Similarly, although not always possible, a configuration that is free of IP addresses means that the configuration can be broadly used, such as reused across pods or in the installation of new data centers.

Summary

This chapter fired the first shots at making BGP configuration more automation friendly. First, we used routing policy to replace IP addresses from individual network statements with a single `redistribute connected` directive with a route map that ensures that

only the appropriate addresses are advertised. Next, building on the small number of addresses covered by /30 and /31 subnets (which makes it easy to determine the remote end's IP address once the local end's IP address is known), we reduce the configuration to use interface names instead of IP addresses to identify a peer.

However, we're not yet done. What this configuration hides is that interfaces still need IP address configuration—even if they're hidden from the BGP configuration and not duplicated. Also, the configuration still relies on knowledge of the peer's ASN. In Chapter 4, we eliminate both of these requirements.

CHAPTER 4
Reimagining BGP Configuration

This chapter shows how router configuration can be reduced by completely eliminating interface IP addresses and specifying the remote-as of each neighbor. Both of these improvements will make configuring a BGP router in the data center a snap, and automation a breeze.

In Chapter 3, we showed how you could eliminate IP address usage from the BGP configuration. However, the operator still needs to configure IP addresses on the interfaces for BGP peering. Because these interface addresses are never used for anything but BGP configuration and their information is never propagated via BGP, their configuration is a meaningless holdover from the service provider world in the data center. Another issue mentioned toward the end of Chapter 3 about automating the configuration is the need to know the remote-as of the peer.

After we eliminate these two requirements, we're left with a configuration that is homogeneous and duplication-free across the nodes, with the only node-specific content being the node's ASN and its `router-id`. In other words, the configuration is very automation friendly, and simple.

To achieve these goals, we'll need to understand a topic almost as old as routing: unnumbered interfaces, and how we adapt this construct to BGP.

The Need for Interface IP Addresses and remote-as

Because BGP runs on TCP/IP, it needs an IP address to create a connection. How can we identify this remote node's address while at the same time not allocating any IP addresses on interfaces? Answering this question will involve understanding a lesser-known RFC and the stateless configuration tools provided by IPv6. It also involves understanding the real heart of routing.

The second problem is that every BGP configuration relies on knowing the remote ASN. But this ASN is really required for only one thing: to identify whether the session is governed by the rules of internal BGP (iBGP) or external BGP (eBGP).

The Numbers on Numbered Interfaces

Is configuring IP addresses on an interface really that big of a deal? How many of them can there be anyway?

Consider a simple two-tier Clos with 4 spines and 32 leaves—a fairly common network. Each spine has 32 links, one to each leaf, and there are 4 spines. This requires 4 * 32 * 2 = 256 IP addresses (4 spines * 32 interfaces * 2 addresses per interface, one for each end). If the number of leaves were to become 96 instead of 32—again not uncommon in mid-sized networks—the total number of interface IP addresses we'd need would be 4 * 96 * 2 = 768. As we increase the scale, say to 16 spines, the total number of addresses would rise to 16 * 96 * 2 = 3,072.

Although deriving these numbers algorithmically is possible, it can be clunky and error prone. The automation code becomes trickier. A very common approach people take is to store the interface addresses as a list or group of variables, and in the automation program, read from these variable sto assign the addresses to interfaces. This method becomes impossible to use.

The sad part of all this is that these addresses are not used for anything but BGP sessions. So why not get rid of them entirely?

Philosophical Aside on Numbered Interfaces

Assigning an IP address to each addressable interface endpoint is a fairly fundamental practice in a traditional Layer 3 (L3) design. But this design leaves the question of who an IP address belongs to: the interface or the node?

One practical question implied by this identity confusion is, "Can a node respond to an Address Resolution Protocol (ARP) request received on an interface for an IP address that is assigned to the node but not assigned to that particular interface?" Routers answered that question with a resounding "No." If you want to enable such behavior on a router, you need to enable a feature called "proxy-arp." Linux answered the same question with a resounding "Yes." The reasoning of the Linux implementers was that they wanted to enable communication to the maximum extent possible. So, the node is free to respond to an ARP request for any IP address it owns, no matter which interface on which the ARP request is received.

The design of Internet Control Message Protocol (ICMP) further cemented the idea that interfaces needed IP addresses. ICMP reports only the IP address of the endpoint where packet forwarding failed. It does not, for example, report the DNS name of the endpoint. Why does this matter, you ask? Traceroute. Traceroute is an old, powerful, and popular tool that people use to debug connectivity problems in the network. If the ICMP response reports the interface's IP address, it is possible to identify not only the node, but also the incoming interface on which the poor packet was rejected. This information then can be used to find the root cause for the lack of connectivity. One of the most frequent questions I am asked is whether traceroute works with unnumbered interfaces (yes, it does, and you can see it for yourself by using the code posted on GitHub).

Finally, ensuring that the two ends of an interface were assigned addresses from the same subnet could be a poor man's way to verify proper cabling.

Unnumbered Interfaces

Early network architects also had explored the other fork in this design decision: not assigning a unique IP address to every interface of a node. An interface without an IP address of its own was called an "unnumbered" interface.

It is not that the interface doesn't have an IP address; it borrows its IP address from another interface. But if the interface from which the IP address is borrowed fails, its IP address can no longer be borrowed. To avoid having interfaces suddenly lose their IP addresses, interfaces borrow the IP address from an interface that never fails: the loopback interface.

Routers can respond to ARPs on unnumbered interfaces with the received interface's local MAC address because the interface has an IP address, even if borrowed. ICMP, with traceroute, still works. But, if an IP address is no longer unique on a node, don't we lose the ability to identify the interface on which the packet entered a router?

Clos networks are predominantly built with just a single link between each pair of nodes. So, it is trivial to identify the links between nodes and thus derive the identity of either the incoming interface or the outgoing interface. If a Clos network does have multiple parallel links between nodes, it is difficult to identify the specific interface among the parallel links at the root of a connectivity issue. However, multiple parallel links between nodes in a Clos network is not common due to various reasons, which are discussed in Chapter 1.

So how do routing protocols deal with unnumbered interfaces? OSPF, which runs over IP, works fine. The original OSPF RFC provided enough guidance on how to make this scenario work. Even though most vendors don't implement it, the open source routing suite FRRouting supports the same practice. Unnumbered OSPF is deployed in production at many sites. IS-IS, which does not even run on IP, also works fine with unnumbered interfaces.

BGP Unnumbered

All of this is well and good, but how can BGP work in a world without interface IP addresses?

In the routing protocol world, there is a chicken-and-egg problem. If the routing protocol is how you advertise reachability to a route, how does a routing protocol itself know how to reach its peer? Many protocols solve this problem by relying on a link-specific multicast address (the multicast is restricted to be distributed only on the link). BGP cannot do this because BGP relies on TCP, which requires unicast packets, not multicast. BGP's solution is to use a shared subnet across the links of the interface connecting the routers.

NOTE

Remember that routing is required only if the destination IP address is in a different subnet from the source IP address. For example, in a 10.0.0.0/24 subnet, traffic within the same subnet, say 10.0.0.1 and 10.0.0.10, will flow without requiring any further routing configuration. IP-connected systems use the ARP protocol to determine reachability within a subnet. A packet from 10.0.0.1 to 10.0.0.10 won't require routing, but a packet from 10.0.0.1 to 10.0.1.1 will. The route for the 10.0.0.0/24 on the interface is called a connected route because the subnet is assumed to be directly reachable (or connected) on that link.

Returning to how BGP peers manage to communicate, traditional eBGP configurations have used the connected route on an interface to reach a neighbor without further configuration. If the peer's IP address is not reachable via a connected subnet, the router doesn't know how to reach the peer's IP address without further configuration (or by running another routing protocol that announces that the address). For example, if every node was assigned only a /32 IP address (where /32 implies that the node is the only entity in that network), BGP would be unable to communicate with the peer. To reach the peer's address, a route for that explicit /32 is needed. Such an additional configuration places further undue burden on the user. This statically configured route is on the peers of the node, which means the user must know which port on each node the peer's route is on to configure the static map.

BGP has some other options, such as using dynamic neighbors (which we touch upon in Chapter 6), but none of them simplify configuration in a meaningful way for the user.

So, how can we, without user configuration and using interface addresses, discover the peer's IP address?

Enter IPv6, and an obscure standard, RFC 5549.

IPv6 Router Advertisement

The IPv6 architects designed IPv6 to work as much as possible without explicit configuration. To this end, every link in an IPv6 network is automatically assigned an IP address that is unique only to that link. Such an address is called the *link local IPv6 address*. The link local address (LLA) is guaranteed to be reachable only by directly connected peers, and only on that interface. Typically, an LLA is derived from the MAC address on the link.

To ensure that hosts automatically discover neighboring routers, a new link-level protocol called router advertisement (RA) was introduced. When enabled on an interface, RA periodically announces the interface's IPv6 addresses, including the LLA. Thus, one end can automatically determine the other end's IPv6 address.

Both IPv6 and RA are universally implemented these days on both hosts and routers. So, this seems like a step in the right direction of making peer addresses automatically discoverable.

To be clear, the use of IPv6 LLA does not require operators to begin deploying IPv6 in their networks. There is also no tunneling of any sort involved, IPv4 in IPv6 or any other, in what we're attempting to use here. The IPv6 LLA is used only to establish a TCP connection for starting a BGP session. Besides enabling IPv6 on a link, which is typically enabled automatically, and the enabling of the IPv6 router advertisement on the link, no other knowledge of IPv6 is expected of the operator.

Even though the peer's IP address has been automatically discovered and a BGP session can be established, this isn't enough to achieve a completely working network.

RFC 5549

Even though we now potentially can establish a BGP peering without requiring an interface IP address, advertising routes also requires a way to specify how to reach the router advertising the routes. In BGP, this is signaled explicitly in the route advertisement via the NEXTHOP attribute. The previous section showed how this

could work together with RA to establish a BGP session over IPv6. We can achieve our unnumbered interface goal if an IPv4 route can use an IPv6 address as the next hop.

As explained in Chapter 1, BGP is a multiprotocol routing suite and allows advertisements and withdrawals of multiple address families to be carried over a single connection. Thus, BGP IPv4 UPDATE messages can be transported over an IPv6 TCP connection, just like IPv6 UPDATE messages can be transported over an IPv4 TCP connection. Advertising IPv4 or IPv6 routes in this case, does not involve any form of tunneling, automatic or otherwise.

In the UPDATE message advertising reachability to routes, BGP includes the nexthop IP address associated with the routes being announced. In the case of IPv4, this is carried as the NEXTHOP attribute in the main attributes section of a BGP UPDATE message (attributes are like Post-it notes that provide additional information about the route being advertised). The nexthop address is of the same family as the route itself. In other words, IPv4 routes are announced with IPv4 nexthops and IPv6 routes are announced with IPv6 nexthops. When carrying an IPv4 route on an eBGP session on an interface without an IPv4 address, what is the nexthop IP address to announce? The only address available on that interface is the IPv6 LLA. Enter RFC 5549.

RFC 5549 is a somewhat obscure RFC, invented in the early years of a new century. Its purpose is to allow the advertisement of an IPv4 route and routing of an IPv4 packet over a pure IPv6 network. Thus, it provides a way to carry IPv4 routes with an IPv6 nexthop. You read that right: IPv4 routes with a nexthop that is an IPv6 address.

Here's a quick recap of how routing works to understand this. Imagine that the route entry for 10.1.1.0/24 is with a nexthop of 20.1.1.1/30 and an outgoing interface of `swp1`.

1. On receiving a packet destined to 10.1.1.1, routing uses this route entry and decides that the nexthop's IP address is 20.1.1.1/30, and that this is our device `swp1`.
2. To deliver the packet to 20.1.1.1, the router needs 20.1.1.1's corresponding MAC address. If the router does not have an ARP entry for 20.1.1.1 in its ARP cache, it runs arp to get the MAC address of 20.1.1.1 on interface `swp1`.

3. The ARP reply from the neighboring router populates the ARP cache with the MAC address of 20.1.1.1 on interface `swp1`.
4. The router then sticks this MAC address as the destination MAC address on the packet, with the source MAC address of interface `swp1`, and sends the packet on its merry way.

Except for getting the MAC address to put on the packet, the nexthop IP address is not used in the packet at all.

In case of IPv6, as well, the nexthop IPv6 address is used to identify the nexthop MAC address, using IPv6's equivalent of ARP: Neighbor Discovery (ND). Even in IPv6, forwarding to the original destination involves only the nexthop's MAC address. The nexthop IP address is used only to get the nexthop's MAC address.

RFC 5549 builds on this observation and provides an encoding scheme to allow a router to advertise IPv4 routes with an IPv6 nexthop.

Forwarding with RFC 5549

But, wait, you say, astute reader. The routing table itself is structured around the assumption that each IPv4 route has an IPv4 nexthop, whereas an IPv6 route has an IPv6 nexthop. RFC 5549 itself doesn't do anything except allow you to work around a BGP issue. Continuing further, you say on a roll, won't this require that IPv4 route forwarding reach into the IPv6 part of the stack, breaking layering, protocol isolation, and goodness knows what else? Won't the solution require hardware support, given that the hardware does pretty much what a software implementation does in routing packets?

A naive implementation would indeed require all that. But then, one does need not be so naive. Although RFC 5549 has been implemented in a few traditional routers, access to the open source FRRouting suite allows us to examine closer how a non-naive implementation works.

FRRouting implements IPv6 RA natively. IPv6 RA has an option to carry the sender's MAC address, as well. FRRouting uses this option to announce its own LLA and MAC address. On receiving an RA packet, the neighboring node's RA code in FRRouting gets the MAC address and the associated IPv6 LLA. Now that the interface's peering address is known, FRRouting kicks BGP into action to start con-

nection establishment. This is also shown by the packet exchange timeline diagram in Figure 4-1.

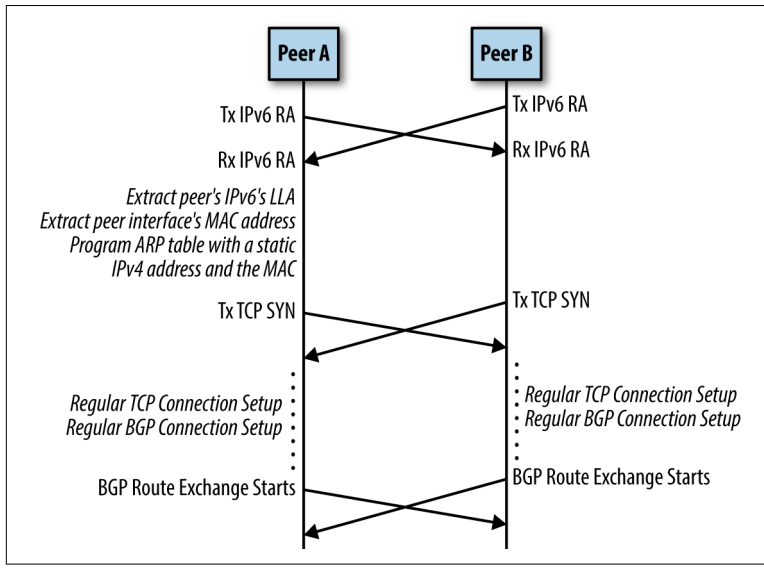


Figure 4-1. BGP unnumbered packet timeline sequence

After a connection has been successfully established, BGP receives a route advertisement for the aforementioned 10.1.1.0/24 from the peer with the peer's IPv6 LLA (and global IPv6 address if one is configured). If BGP selects this path as the best path to reach 10.1.1.0/24, it passes this route down to the Routing Information dataBase (RIB) process (called zebra in FRRouting), with the nexthop set to the IPv6 LLA, this nexthop information being received in the BGP UPDATE message.

NOTE

RIB is a collection of all routes received from every routing protocol running on the node and statically configured routes. If there are multiple announcers for a route, the RIB process picks one with the lowest value of a field called distance. There are default values for distance for each protocol, but the user can change them, as well.

On receiving a route for 10.1.1.0/24 with an IPv6 LLA, assume that the RIB picks this as the best route with which to populate the forwarding table. The RIB process now consults its database to see

whether it has the information for the MAC address associated with this IPv6 LLA. Let this MAC address be 00:00:01:02:03:04. The RIB process now adds a static ARP entry for 169.254.0.1 with this MAC address, pointing out the peering interface. 169.254.0.1 is an IPv4 LLA, although it is not automatically assigned to an interface the way IPv6 LLA is. FRRouting assumes that 169.254.0.1 is reserved (as of this writing, this cannot be changed through a configuration option). The reason for the static ARP entry is so that the router cannot run ARP to get this address; this IP address was assigned by the router implicitly without its neighbor knowing anything about this assignment; thus, the neighbor cannot respond to the ARP, because it doesn't have the IP address assigned to the interface.

The RIB process then pushes the route into the kernel routing table with a nexthop of 169.254.0.1 and an outgoing interface set to that of the peering interface. So, the final state in the tables looks like this:

```
ROUTE: 10.1.1.0/24 via 169.254.0.1 dev swp1
ARP: 169.254.0.1 dev swp1 lladdr 00:00:01:02:03:04 PERMANENT
```

At this point, everything is set up for packet forwarding to work correctly. More specifically, the packet forwarding logic remains unchanged with this model.

If the link goes down or the remote end stops generating an RA, the local RA process yanks out the LLA and its associated MAC from the RIB. This causes the RIB process to decide that the nexthop is no longer reachable, which causes it to notify the BGP process that the peer is no longer reachable. RIB also tears down the static ARP entry that it created. Terminating the session causes BGP to yank out the routes pointing out this peering interface.

To summarize:

- BGP unnumbered uses the interface's IPv6 LLA to set up a BGP session with a peer.
- The IPv6 LLA of the remote end is discovered via IPv6's Router Advertisement (RA) protocol.
- RA provides not only the remote end's LLA, but also its corresponding MAC address.
- BGP uses RFC 5549 to encode IPv4 routes as reachable over an IPv6 nexthop, using the IPv6 LLA as the nexthop.

- The RIB process programs a static ARP entry with a reserved IPv4 LLA, 169.254.0.1, with the MAC address set to the one learned via RA.
- BGP hands down to the RIB process IPv4 routes with the IPv6 LLA as the nexthop.
- The RIB process converts the nexthop to 169.254.0.1 and the outgoing interface before programming the route in the forwarding table.

BGP Capability to Negotiate RFC 5549 Use

Because encoding IPv4 routes with an IPv6 nexthop is not the usual model, RFC 5549 defines a new capability, called *extended nexthop*, to negotiate the use of RFC 5549 over a peering session. As is common with BGP capabilities, both sides must advertise their capability to understand RFC 5549 in order for it to be used in the BGP peering.

FRRouting automatically enables RA on an interface and enables the sending of the extended nexthop BGP capability, when a BGP peering is set up to be based on an interface that does not have an IPv4 address.

Interoperability

Every eBGP peer sets the NEXTHOP to its own IP address before sending out a route advertisement.

Figure 4-2 shows a hypothetical network in which routers B and D support RFC 5549, whereas routers A and C do not. So, there are interface IP addresses on the links between B and A and between B and C. When A announces reachability to 10.1.1.0/24, it provides its peering interface's IPv4 address as the nexthop. When B advertises reachability to 10.1.1.0/24, it sets its IPv6 LLA as the nexthop when sending the route to D, and sets its interface's IPv4 address as the nexthop when sending the route to C.

In the reverse direction, if D announces reachability to a prefix 10.1.2.0/24, it uses its interface's IPv6 LLA to send it to B. When B announces this to A and C, it sets the nexthop to be that of the IPv4 address of the peering interface.

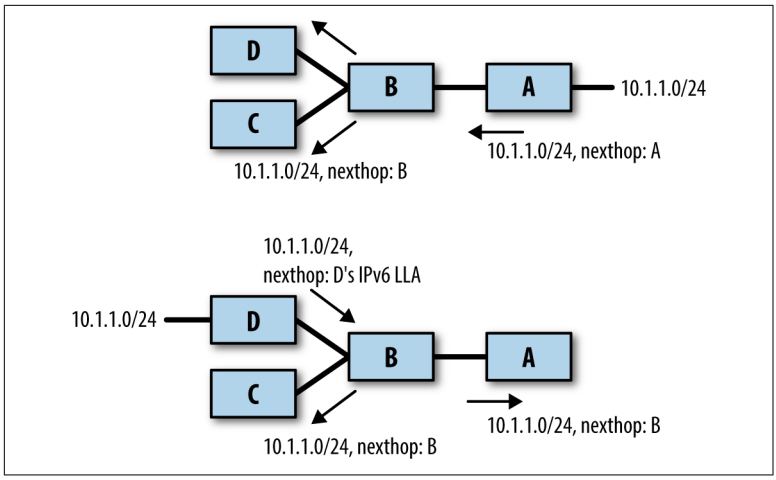


Figure 4-2. Interoperability with RFC 5549

A remote-as By Any Other Name

After eliminating interface addresses, the only thing remaining to accomplish the goal of the simple, cookie-cutter configuration is the need to specify the neighbor's ASN via the `remote-as` keyword of a BGP neighbor configuration.

There are two primary uses for specifying neighbor's ASN in the neighbor specification:

- In the spirit of connecting across administrative domains, and where harm on a large financial and global scale is possible by connecting to the wrong administrative domain accidentally, it is critical to verify operator intent.
- To identify whether the BGP session will be governed by iBGP rules or eBGP rules.

Within the data center, because we're not crossing administrative domains, security is no longer a compelling reason to specify the ASN. And, if the only reason is to identify what rules govern the session, that can be done by a simple non-neighbor-specific field.

Based on this reasoning, FRRouting added two new choices to the remote-as keyword: `external` and `internal`. "External" means that you expect to set up an eBGP connection with this neighbor, whereas "internal" means that you expect to set up an iBGP connec-

tion. In reality, you can even ignore this specification because you can identify iBGP versus eBGP by the ASN received in the BGP OPEN message. However, the remote-as command helps kick off creation of the BGP peer data structure, as it's easy to make a typo in the neighbor specification in one of the commands and accidentally create a new BGP peer. For example, if there were a peer169.254.1.11 and there was a typo in one of the neighbor commands—`neighbor 169.254.11.1 timers connect 9` instead of `neighbor 169.254.1.11 timers connect 9`—you don't want BGP to begin spinning up a new neighbor session.

Summary

By eliminating interface IP addresses and the specification of the exact remote-as in the neighbor command specification, we can arrive at a configuration, listed in Example 4-1, that looks remarkably similar across the leaves and spines illustrated in Figure 3-1. The only differences between the nodes are shown in bold in the example.

Example 4-1. Final BGP configuration for a leaf and spine in a Clos network

```
// leaf01 configuration

log file /var/log/frr/frr.log
ip prefix-list DC_LOCAL_SUBNET 5 permit 10.1.0.0/16 le 26
ip prefix-list DC_LOCAL_SUBNET 10 permit 10.0.254.0/24 le 32
route-map ACCEPT_DC_LOCAL permit 10
  match ip-address DC_LOCAL_SUBNET

router bgp 65000
  bgp router-id 10.0.254.1
  neighbor peer-group ISL
  neighbor ISL remote-as external
  neighbor swp51 interface peer-group ISL
  neighbor swp52 interface peer-group ISL
  address-family ipv4 unicast
    neighbor ISL activate
    redistribute connected route-map ACCEPT_DC_LOCAL

// spine01 configuration

log file /var/log/frr/frr.log
ip prefix-list DC_LOCAL_SUBNET 5 permit 10.1.0.0/16 le 26
ip prefix-list DC_LOCAL_SUBNET 10 permit 10.0.254.0/24 le 32
```

```
route-map ACCEPT_DC_LOCAL permit 10
  match ip-address DC_LOCAL_SUBNET

router bgp 65534
  bgp router-id 10.0.254.254
  neighbor peer-group ISL
  neighbor ISL remote-as external
  neighbor swp1 interface peer-group ISL
  neighbor swp2 interface peer-group ISL
  neighbor swp3 interface peer-group ISL
  neighbor swp4 interface peer-group ISL
  address-family ipv4 unicast
    neighbor ISL activate
    redistribute connected route-map ACCEPT_DC_LOCAL
```

This is a far cry from the original node-specific BGP configuration. The configuration is also extremely trivial to automate using tools such as Ansible, Puppet, or Chef. This is due not only to the elimination of just about every router-specific information via the use of interface names, but also, more important, each router's configuration contains information that is completely local to the router, with no information about the peer.

We've so far focused on configuring BGP in a Clos topology. We have not described how to view the results of our configuration, manage BGP after the initial configuration, or how to configure BGP to connect a Clos topology to the external world. These are the focus of Chapter 5.

CHAPTER 5

BGP Life Cycle Management

So far, this book has laid the groundwork to create a simple, automatable configuration for a data center network using BGP. But we have just gone through the initial configuration of a leaf or spine router. As any network operator knows, the job is far from done after the network is deployed. Routers need to be upgraded, security patches need to be applied, new routers need to be rolled in, and heaven help us all, what if BGP refuses to behave? This chapter addresses these questions.

Useful show Commands

So far, we've only discussed configuring BGP, without viewing the fruits of our labor. This section covers two of the most useful and common commands used to view BGP's status. This section is intended to help network operators new to BGP get off the ground (though some old masters might learn a thing or two that's new, as well), and to highlight differences from the traditional operation, not be a complete reference. There are lots of online and printed documentation on the various `show` commands used in BGP.

Displaying BGP Session Information

The most popular command to see BGP's status is `show ip bgp summary`. Figure 5-1 shows sample output for the command for the reference topology in this book (based on FRRouting).

```
leaf01# sh ip bgp summary
BGP router identifier 10.254.0.1, local AS number 64513 vrf-id 0
BGP table version 13
RIB entries 15, using 2040 bytes of memory
Peers 2, using 42 KiB of memory
Peer groups 1, using 72 bytes of memory

Neighbor        V         AS MsgRcvd MsgSent   TblVer  InQ OutQ  Up/Down State/PfxRcd
spine01(swp51)  4      65000   26602   26604        0    0    0 22:09:51            5
spine02(swp52)  4      65000   26601   26603        0    0    0 22:09:51            5

Total number of neighbors 2
leaf01#
```

Figure 5-1. Showing the network

This command shows only the output of IPv4 BGP sessions. When BGP began life, there was only IPv4 and the keyword `ip` was unambiguous with respect to what protocol it referred to. Since the advent of IPv6, and with the evolution of BGP to support multiple protocols, we need a command to display IPv6 sessions, as well. In line with the AFI/SAFI model, the `show bgp` commands have evolved to support `show bgp ipv4 unicast summary` and `show bgp ipv6 unicast summary`. For many operators, however, sheer muscle memory forces them to type `show ip bgp summary`.

Following are the key points to note in this output:

- All the neighbors with whom this router is supposed to peer are listed (unlike with other protocols such as OSPF).
- The state of each session is listed. If a session is in the Established state, instead of the state name, the number of prefixes accepted from the peer is shown.
- Every session's uptime is shown (or its downtime, if the session is not in Established state).
- Information such as the node's router ID and ASN is also shown.

The version of the BGP (the "V" column in Figure 5-1) is archaic, given that all BGP implementations in use today, especially in the data center, are running version 4 of the protocol. The remaining fields are mostly uninteresting unless there's a problem.

One difference to note in the previous output compared to what you might see in just about every other implementation (except ExaBGP) is the display of the hostname of the peer. This is based on an IETF draft that defined a new BGP capability, called *hostname*, which allows operators to advertise the hostname along with the

BGP open message. This makes for simpler debugging because it is easier to remember hostnames than interface names. The Internet Assigned Numbers Authority (IANA) has issued a standard capability ID to be used for this capability.

So, any `show` or `clear` command can take a hostname instead of a neighbor. The use of hostnames in commands potentially simplifies troubleshooting, because it is far more intuitive to say "show me the state of my session with host *x*" than "show me the state of my session with IP address *x*," or "show me the state of my session on interface *x*."

In FRRouting, we can use hostnames in any command that doesn't configure a BGP session. The reason for the restriction is that the hostname is not known until BGP capabilities are negotiated and the hostnames exchanged.

For detailed information about a neighbor, you can use the command `show ip bgp neighbors` *`neighbor_name`*. The output of this command contains additional information, such as the last time the session was reset, the reason for the reset, and the number of BGP UPDATE messages sent and received. Example 5-1 presents a sample output from leaf01 for the neighbor spine01.

Example 5-1. Sample output showing details of BGP peering session with a neighbor

```
BGP neighbor on swp51: fe80::4638:39ff:fe00:5c, remote AS 65000,
local AS 64513, external link
Hostname: spine01
 Member of peer-group fabric for session parameters
  BGP version 4, remote router ID 10.254.0.254
  BGP state = Established, up for 00:02:36
  Last read 00:00:00, Last write 00:02:35
  Hold time is 9, keepalive interval is 3 seconds
  Neighbor capabilities:
    4 Byte AS: advertised and received
    AddPath:
      IPv4 Unicast: RX advertised IPv4 Unicast and received
      IPv6 Unicast: RX advertised IPv6 Unicast and received
    Extended nexthop: advertised and received
      Address families by peer:
                  IPv4 Unicast
    Route refresh: advertised and received(old & new)
    Address family IPv4 Unicast: advertised and received
    Address family IPv6 Unicast: advertised and received
    Hostname Capability: advertised and received
```

```
    Graceful Restart Capabilty: advertised and received
      Remote Restart timer is 120 seconds
      Address families by peer:
        none
  Graceful restart informations:
    End-of-RIB send: IPv4 Unicast, IPv6 Unicast
    End-of-RIB received: IPv4 Unicast, IPv6 Unicast
  Message statistics:
    Inq depth is 0
    Outq depth is 0
                         Sent       Rcvd
    Opens:                  4          5
    Notifications:          4          2
    Updates:               40         25
    Keepalives:         26962      26959
    Route Refresh:          0          0
    Capability:             0          0
    Total:              27010      26991
  Minimum time between advertisement runs is 0 seconds

 For address family: IPv4 Unicast
  fabric peer-group member
  Update group 1, subgroup 1
  Packet Queue length 0
  Community attribute sent to this neighbor(both)
  5 accepted prefixes

 For address family: IPv6 Unicast
  fabric peer-group member
  Update group 2, subgroup 2
    End-of-RIB send: IPv4 Unicast, IPv6 Unicast
    End-of-RIB received: IPv4 Unicast, IPv6 Unicast
  Message statistics:
    Inq depth is 0
    Outq depth is 0
                         Sent       Rcvd
    Opens:                  4          5
    Notifications:          4          2
    Updates:               40         25
    Keepalives:         26988      26985
    Route Refresh:          0          0
    Capability:             0          0
    Total:              27036      27017
  Minimum time between advertisement runs is 0 seconds

 For address family: IPv4 Unicast
  fabric peer-group member
  Update group 1, subgroup 1
  Packet Queue length 0
  Community attribute sent to this neighbor(both)
  5 accepted prefixes
```

```
 For address family: IPv6 Unicast
  fabric peer-group member
  Update group 2, subgroup 2
  Packet Queue length 0
  Community attribute sent to this neighbor(both)
  0 accepted prefixes

  Connections established 3; dropped 2
  Last reset 00:03:57, due to NOTIFICATION sent (Cease/Connection
  collision resolution)
  Message received that caused BGP to send a NOTIFICATION:
    FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
    00660104 FDE80009 0AFE00FE 49020601
    04000100 01020601 04000200 01020805
    06000100 01000202 02800002 02020002
    06410400 00FDE802 0A450800 01010100
    02010102 0B490907 7370696E 65303100
    02044002 0078
Local host: fe80::4638:39ff:fe00:5b, Local port: 179
Foreign host: fe80::4638:39ff:fe00:5c, Foreign port: 52191
Nexthop: 10.254.0.1
Nexthop global: fe80::4638:39ff:fe00:5b
Nexthop local: fe80::4638:39ff:fe00:5b
BGP connection: shared network
BGP Connect Retry Timer in Seconds: 10
Read thread: on  Write thread: off
```

Displaying Routes Exchanged

Another common command is to see the list of routes computed and in BGP's routing table. The command for this is `show ip bgp` or `show bgp ipv4 unicast`. Figure 5-2 shows the output of this command on leaf01 of the reference topology.

The key fields are the prefix itself, the possible nexthops, and the AS_PATH associated with each prefix. This screen displays only 8 out of 12 prefixes because the other two were not accepted. Frequent causes for rejecting prefixes are either policy decisions or because an AS_PATH loop was detected (AS_PATH loop was detected in this case). The asterisk (*) at the beginning of a line indicates that the route is valid; that is the nexthop is reachable. Following that, an equal sign (=) indicates that the route has multiple usable equal-cost paths.

```
leaf01# sh ip bgp
BGP table version is 13, local router ID is 10.254.0.1
Status codes: s suppressed, d damped, h history, * valid, > best, = multipath,
              i internal, r RIB-failure, S Stale, R Removed
Origin codes: i - IGP, e - EGP, ? - incomplete

   Network          Next Hop            Metric LocPrf Weight Path
*> 10.1.20.0/24     0.0.0.0                  0         32768 i
*> 10.3.20.0/24     swp52                         0 65000 64515 i
*=                  swp51                         0 65000 64515 i
*> 10.254.0.1/32    0.0.0.0                  0         32768 i
*> 10.254.0.2/32    swp52                         0 65000 64514 i
*=                  swp51                         0 65000 64514 i
*> 10.254.0.3/32    swp52                         0 65000 64515 i
*=                  swp51                         0 65000 64515 i
*> 10.254.0.4/32    swp52                         0 65000 64516 i
*=                  swp51                         0 65000 64516 i
*> 10.254.0.253/32  swp52           0             0 65000 i
*> 10.254.0.254/32  swp51           0             0 65000 i

Displayed  8 out of 12 total prefixes
leaf01#
```

Figure 5-2. BGP routes as seen on leaf01 of the reference topology

You can use the same command with a specific prefix to get the details of the received prefix advertisement. For example, Figure 5-3 depicts the output of the command `show ip bgp 10.254.0.3`.

```
leaf01# sh ip bgp 10.254.0.3
BGP routing table entry for 10.254.0.3/32
Paths: (2 available, best #1, table Default-IP-Routing-Table)
  Advertised to non peer-group peers:
  spine01(swp51) spine02(swp52)
  65000 64515
    fe80::4638:39ff:fe00:2b from spine02(swp52) (10.254.0.253)
    (fe80::4638:39ff:fe00:2b) (used)
      Origin IGP, localpref 100, valid, external, multipath, bestpath-from-AS 65000, best
      AddPath ID: RX 0, TX 12
      Last update: Fri May 19 16:44:04 2017

  65000 64515
    fe80::4638:39ff:fe00:5c from spine01(swp51) (10.254.0.254)
    (fe80::4638:39ff:fe00:5c) (used)
      Origin IGP, localpref 100, valid, external, multipath
      AddPath ID: RX 0, TX 8
      Last update: Fri May 19 16:44:04 2017

leaf01#
```

Figure 5-3. Advertised prefix

Using this, a network operator can examine closely to determine details about a prefix, such as what attributes were received for a prefix, and to whom else the prefix was advertised.

Connecting to the Outside World

One of the things we haven't discussed is how we advertise the routes outside the data center. This task also typically falls under the purview of a data center network operator.

Let's use the reference topology we've used throughout this book, as presented in Figure 5-4.

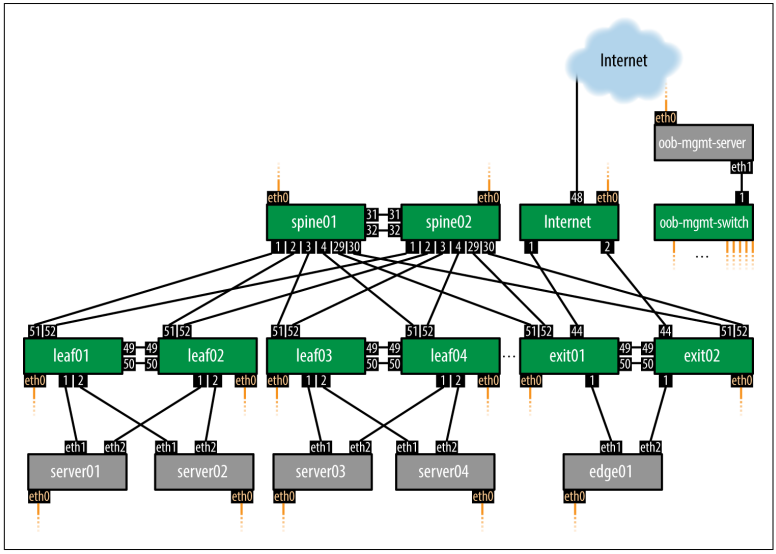


Figure 5-4. Reference topology used in this book

exit01 and exit02 are the two nodes that demarcate the inside of the data center from the outside. They're connected to the node titled `internet`; this is the data center's edge switch, which is the switch that peers with the external world. exit01 and exit02 are called *border leaves* or *exit leaves* (the border leaves maybe in a border pod in a three-tier Clos network as described in Chapter 1).

Border leaves serve two primary functions: stripping off the private ASNs, and optionally aggregating the internal data center routes and announcing only the summary routes to the edge routers.

You strip the private ASNs from the path via the command `neighbor` *`neighbor_name`* `remove-private-AS all`.

You can summarize routes and announce only the aggregate via the command `aggregate-address` *`summary-route`* `summary-only`.

The keyword `summary-only` specifies that the individual routes must not be sent. Without that option, summary routes as well as individual routes are advertised. When a route is aggregated and only the summary route announced, the entire AS_PATH is also removed unless specified otherwise.

Scheduling Node Maintenance

The life cycle of a router typically involves upgrading the software. The upgrade might cover the entire router, just the routing software, or other relevant software that causes the router to lose its peering session with the neighbors as it restarts. If a router's neighbors continue to forward traffic while the router restarts, traffic can be dropped and cause unnecessary traffic loss. To avoid this, especially when the operator knows that the node is going to be taken down, it is useful to allow the neighbors to route around the router. For example, if spine01 is going to be upgraded, you should ask all the leaves to ignore spine01 in their best path computation and send all traffic to only spine02 during this time to ensure a smooth traffic flow. Similarly, in the case of the leaves with dual-attached servers, it would be useful for the spines to avoid sending traffic to the leaf undergoing the upgrade and use only the working leaf. In this fashion, routers can be upgraded, one box at a time, without causing unnecessary loss of traffic.

As discussed in Chapter 1, a modern data center has more than two spine nodes, with four being the most common especially in medium-to-large enterprises. With four nodes, when a spine is taken out of service for maintenance, the network can keep humming along at 75 percent capacity. In a traditional enterprise network design, there are only two spine nodes, which would result in a more significant loss of capacity when a single spine is taken out of service. It is true that the servers would operate at only half their capacity if they were dual-attached. This is why some large enterprises use dual-attached servers only for failover, not with both links active at the same time. Web-scale data centers address this issue by only singly connecting servers, and having so many racks that taking down a single rack is not a big deal. These super-large networks also operate with 16 or 32 spines and so the loss of a single spine results in a drop of just 1/16 or 1/32 of the inter-switch capacity.

The most common and interoperable way to drain traffic is to force the routes to be advertised from the node with an additional ASN added to the advertisement, causing the AS_PATH length to increase in comparison to the node's peers. For example, a route advertised by leaf01 is seen by leaf03 as having multiple paths, one via spine01 and the other via spine02, both with AS_PATH length of

2. If we want to upgrade spine02, we can increase its AS_PATH length, and leaf03 will stop using spine02 to reach leaf01.

Typically, the node's own ASN is used to prepend additional ASNs. Here is an example of a configuration snippet on spine01 prepending its own ASN in its announcements to all neighbors:

```
route-map SCHED_MAINT permit 10
   set as-path prepend 65000 65000

neighbor ISL route-map SCHED_MAINT out
```

Figure 5-5 shows the output for the same prefix used in Figure 5-4, except that one of the spines, spine02, has announced a path that is longer than the other one, and as a result that path has not been selected.

There are other methods to indicate that a BGP router is failing, but not all implementations support these methods, and so I have chosen to talk about the most supported model.

```
leaf01# sh ip bgp 10.254.0.3
BGP routing table entry for 10.254.0.3/32
Paths: (2 available, best #2, table Default-IP-Routing-Table)
  Advertised to non peer-group peers:
  spine01(swp51) spine02(swp52)
  65000 65000 65000 64515
    fe80::4638:39ff:fe00:2b from spine02(swp52) (10.254.0.253)
    (fe80::4638:39ff:fe00:2b) (used)
      Origin IGP, localpref 100, valid, external
      AddPath ID: RX 0, TX 12
      Last update: Sat May 20 15:01:25 2017

  65000 64515
    fe80::4638:39ff:fe00:5c from spine01(swp51) (10.254.0.254)
    (fe80::4638:39ff:fe00:5c) (used)
      Origin IGP, localpref 100, valid, external, bestpath-from-AS 65000, best
      AddPath ID: RX 0, TX 8
      Last update: Fri May 19 16:44:04 2017

leaf01#
```

Figure 5-5. A path not chosen

Debugging BGP

Like any other software, BGP will occasionally behave unpredictably due to a bug or to a misunderstanding by the operator. A common solution to such a problem is to enable debugging and look at the debug logs to determine the cause of the unpredictable behavior.

Different router software provides different knobs to tweak during debugging. In FRRouting, the command `debug bgp` is the gateway

to understanding what's going on with BGP. There are many options listed under `debug`, but three in particular are key:

`neighbor-events`
: This is used to debug any session and bring up issues. The debugging can be for all sessions, or for only a specific session. Information such as which end initiated the connection, the BGP state machine transitions, and what capabilities were exchanged can all be seen in the debug log with this option enabled.

`bestpath`
: This is used to debug bestpath computation. If you enable it for a specific prefix, the logs will show the logic followed in selecting the bestpath for a prefix, including multipath selection. Figure 5-6 shows an example of the snippet from a log. This is for debugging the same prefix shown in Figure 5-3 and Figure 5-5. As seen, you also can use the debug logs to gain a better understanding of how BGP's bestpath selection logic works—in this case, how a longer AS_PATH prevents a path from being selected.

```
BGP: 10.254.0.3/32: Comparing path swp51 flags 0x418 with path swp52 flags 0x410
BGP: 10.254.0.3/32: path swp51 wins over path swp52 due to aspath hopcount 2 < 4
BGP: 10.254.0.3/32: path swp51 is the bestpath from AS 65000
BGP: 10.254.0.3/32: path swp51 is the initial bestpath
BGP: 10.254.0.3/32: After path selection, newbest is path swp51 oldbest was swp51
BGP: 10.254.0.3/32: Comparing path swp52 flags 0x410 with path swp51 flags 0x498
BGP: 10.254.0.3/32: path swp52 loses to path swp51 due to aspath hopcount 4 > 2
BGP: 10.254.0.3/32: path swp51 is the bestpath, add to the multipath list
BGP: 10.254.0.3/32: starting mpath update, newbest swp51 num candidates 1 old-mpath-count 0
BGP: 10.254.0.3/32: comparing candidate swp51 with existing mpath NONE
BGP: 10.254.0.3/32: New mpath count (incl newbest) 1 mpath-change NO
```

Figure 5-6. Sample debug log showing bestpath computation

`Updates`
: This is used to debug problems involving either advertising or receiving advertisements of prefixes with a neighbor. You can specify a single prefix, all prefixes, or all prefixes for a single neighbor in order to more closely examine the root cause of a problem. The debug logs show you not only the prefixes that were accepted, but also the ones that were rejected. For example, given that the spines share the same ASN, the loopback IP address of a spine cannot be seen by the other spines. To see this in action, by issuing `debug bgp updates prefix 10.254.0.253/32`, we get the output shown by Example 5-1 in the log file.

Example 5-2. Prefix rejected because of ASN loop

```
2017/05/20 15:09:54.112100 BGP: swp2 rcvd UPDATE w/ attr: , origin i,
  mp_nexthop fe80::4638:39ff:fe00:2e(fe80::4638:39ff:fe00:2e),
  path 64514 65000 65000 65000 64515
2017/05/20 15:09:54.112165 BGP: swp2 rcvd UPDATE about 10.254.0.3/32
  -- DENIED due to: as-path contains our own AS;
2017/05/20 15:09:54.113438 BGP: swp3 rcvd UPDATE w/ attr: , origin i,
  mp_nexthop fe80::4638:39ff:fe00:57(fe80::4638:39ff:fe00:57),
  metric 0, path 64515
2017/05/20 15:09:54.113471 BGP: swp3 rcvd 10.254.0.3/32
2017/05/20 15:09:54.113859 BGP: swp4 rcvd UPDATE w/ attr: , origin i,
  mp_nexthop fe80::4638:39ff:fe00:43(fe80::4638:39ff:fe00:43),
  path 64516 65000 65000 65000 64515
2017/05/20 15:09:54.113886 BGP: swp4 rcvd UPDATE about 10.254.0.3/32
  -- DENIED due to: as-path contains our own AS;
2017/05/20 15:09:54.114135 BGP: swp1 rcvd UPDATE w/ attr: , origin i,
  mp_nexthop fe80::4638:39ff:fe00:5b(fe80::4638:39ff:fe00:5b),
  path 64513 65000 65000 65000 64515
2017/05/20 15:09:54.114157 BGP: swp1 rcvd UPDATE about 10.254.0.3/32
  -- DENIED due to: as-path contains our own AS;
2017/05/20 15:09:54.162440 BGP: u3:s6 send UPDATE w/ attr: , origin i,
  mp_nexthop ::(::), path 64515
2017/05/20 15:09:54.162788 BGP: u3:s6 send UPDATE 10.254.0.3/32
2017/05/20 15:09:54.214657 BGP: swp4 rcvd UPDATE w/ attr: , origin i,
  mp_nexthop fe80::4638:39ff:fe00:43(fe80::4638:39ff:fe00:43),
  path 64516 65000 64515
2017/05/20 15:09:54.214803 BGP: swp4 rcvd UPDATE about 10.254.0.3/32
  -- DENIED due to: as-path contains our own AS;
2017/05/20 15:09:54.214914 BGP: swp2 rcvd UPDATE w/ attr: , origin i,
  mp_nexthop fe80::4638:39ff:fe00:2e(fe80::4638:39ff:fe00:2e),
  path 64514 65000 64515
2017/05/20 15:09:54.214933 BGP: swp2 rcvd UPDATE about 10.254.0.3/32
  -- DENIED due to: as-path contains our own AS;
2017/05/20 15:09:54.216418 BGP: swp1 rcvd UPDATE w/ attr: , origin i,
  mp_nexthop fe80::4638:39ff:fe00:5b(fe80::4638:39ff:fe00:5b),
  path 64513 65000 64515
2017/05/20 15:09:54.216449 BGP: swp1 rcvd UPDATE about 10.254.0.3/32
  -- DENIED due to: as-path contains our own AS;
```

Summary

This chapter provided information for some of the less frequent, but nevertheless critical tools and tasks for managing and troubleshooting BGP deployments in a data center. At this stage, you should hopefully possess a good understanding of data center networks, BGP, and how to configure and manage a Clos network in the data center.

Chapter 6 covers extending BGP routing all the way to the host, something that is also increasingly being deployed as a solution in the data center due to the rise in virtual services, among other uses.

CHAPTER 6

BGP on the Host

The advent of the modern data center revolutionized just about everything we know about computing and networking. Whether it be the rise of NoSQL databases, new application architectures and microservices, or Clos networks with routing as the fundamental rubric rather than bridging, they have each upended hitherto well-regarded ideas. This also has affected how services such as firewalls and load balancers are deployed.

This chapter examines how the new model of services shifts routing all the way to the server, and how we configure BGP on the host to communicate with the ToR or leaf switch.

Traditional network administrators' jurisdiction ended at the ToR switch. Server administrators handled server configuration and management. In the new-world order, either separate server and network administrators have been replaced by a single all-around data center operator, or network administrators must work in conjunction with server administrators to configure routing on hosts, as well. In either case, it is important for a data center operator to ensure that the configuration of BGP on the host does not compromise the integrity of the network.

The Rise of Virtual Services

In traditional data center networks, the boundary between bridging and routing, the L2–L3 gateway, was where services such as firewall and load balancers were deployed. The boundary was a natural fit

because the boundary represented in some sense the separation of the client from the server. It was logical to assign firewalls at this boundary to protect servers from malicious or unauthorized clients. Similarly, load balancers front-ended servers, typically web servers, in support of a scale-out model. This design also extended to firewalls, where load balancers front-ended a row of firewalls when the traffic bandwidth exceeded the capacity of a single firewall.

These firewalls and load balancers were typically appliances, which were usually scaled with the scale-in model; that is, purchasing larger and larger appliances to support the increasing volume of traffic.

The Clos network destroyed any such natural boundary, and with its sheer scale, the modern data center made scale-in models impractical. In the new world, the services are provided by virtual machines (VMs) running on end hosts or nonvirtualized end hosts. Two popular services provided this way are the load balancer and firewall services. In this model, as the volume of traffic ebbs and flows, VMs can be spun up or down dynamically to handle the changing traffic needs.

Anycast Addresses

Because the servers (or VMs) providing a service can pop up anywhere in the data center, the IP address no longer can be constrained to a single rack or router. Instead, potentially several racks could announce the same IP address. With routing's ECMP forwarding capability, the packets would flow to one of the nearest nodes offering the service. These endpoint IP addresses have no single rack or switch to which they can be associated. These IP addresses that are announced by multiple endpoints are called *anycast* IP addresses. They are unicast IP addresses, meaning that they are sent to a single destination (as opposed to multidestination addresses such as multicast or broadcast), but the destination that is picked is determined by routing, and different endpoints pick different nodes offering the same service.

Subnets are typically assigned per rack. As we discussed in Chapter 1, 40 servers per rack result in the ToR announcing a /26 subnet. But how does a ToR discover or advertise a nonsubnet address that is an anycast service IP address? Static routing configuration is not acceptable. BGP comes to the rescue again.

BGP Models for Peering with Servers

There are two models for peering with servers. The first is the BGP unnumbered model outlined in Chapter 4. The second involves a feature that BGP supports called *dynamic neighbors*. We'll examine each model, listing the pros and cons of both. But we begin by looking at what's common to both models: the ASN numbering scheme, and the route exchange between the server and the ToR.

ASN Assignment

The most common deployment I have seen is to dedicate an ASN for all servers. The advantages of this approach are that it is simple to configure and automate, and it simplifies identifying and filtering routes from the server. The two main disadvantages of this approach are 1) the complexity of the configuration on the server increases if we need to announce anything more than just the default route to the host, and 2) tracking which server announced a route becomes trickier because all servers share the same ASN.

Another approach would be to assign a single ASN for all servers attached to the same switch, but separate ASNs for separate switches. In a modern data center, this translates to having a separate server ASN per rack. The benefit of this model is that it now looks like the servers are just another tier of a Clos network. The main disadvantages of this model are the same as the previous model's, though we can narrow a route announcement to a specific rack.

The final approach is to treat each server as a separate node and assign separate ASNs for each server. Although a few customers I know of are using this approach, it feels like overkill. The primary benefits of this approach are that it perfectly fits the model prescribed for a Clos network, and that it is easy to determine which server advertised a route. Given the sheer number of servers, using 4-byte ASNs seems the prudent thing to do with this approach.

Route Exchange Model

Because each host is now a router of first order, all sorts of bad things can happen if we do not control what routes a switch accepts from a host. For example, a host can accidentally or maliciously announce the default route (or any other route that it does not own), thereby delivering traffic to the wrong destination. Another

thing to guard against is to ensure that the ToR (or leaf) switch never thinks the host is a transit node; that is, one with connectivity to other nodes. That error would result in severe traffic loss because a host is not designed to handle traffic loads of hundreds of gigabits per second. Lastly, the router connected to the server announces only the default route. This is to avoid pushing too many routes to the host, which could fill up its routing table and make the host waste precious cycles trying to run the best path algorithm every time some route changes (for example, when a ToR switch loses connectivity to a leaf or spine switch).

To handle all of these scenarios, we use routing policies as described in Chapter 3. The following configuration snippet shows how we can accomplish each of the aforementioned tasks via the use of routing policy, as demonstrated here:

```
ip prefix-list ANYCAST_VIP seq 5 permit 10.1.1.1/32
ip prefix-list ANYCAST_VIP seq 10 permit 20.5.10.110/32

ip prefix-list DEFONLY seq 5 permit 0.0.0.0/0

route-map ACCEPT_ONLY_ANYCAST permit 10
   match ip address prefix-list ANYCAST_VIP

route-map ADVERTISE_DEFONLY permit 10
   match ip address prefix-list DEFONLY

neighbor server route-map ACCEPT_ONLY_ANYCAST in
neighbor server route-map ADVERTISE_DEFONLY out
neighbor server default-originate
```

In this configuration, the neighbor statement with the `route-map ACCEPT_ONLY_ANYCAST` says that the only route advertisements accepted from a neighbor belonging to the `peer-group` server are the anycast IP addresses listed in the `ANYCAST_VIP prefix-list`. Similarly, the neighbor statement with the `route-map ADVERTISE_DEFONLY` specifies that BGP advertise only the default route to any neighbor belonging to the `peer-group` server.

BGP Peering Schemes for Edge Servers

Now that we have established the importance of including edge servers such as load balancers and firewalls in your routing configuration, we can look at two BGP models for doing so: *dynamic neighbors* and *BGP unnumbered*. Each model has limitations, so look over

the following subsections and decide which comes closest to meeting the needs in your data center.

Dynamic neighbors

Because BGP runs over TCP, as long as one of the peers initiates a connection, the other end can remain passive, silently waiting for a connection to come, just as a web server waits for a connection from a browser or other client.

BGP dynamic neighbors is a feature supported in some implementations whereby one end is typically passive. It is just told what IP subnet to accept connections from, and is associated with a peer group that controls the characteristics of the peering session.

Recall that the servers within a rack typically share a subnet with the other servers in the same rack. As an example, let's assume that a group of 40 servers connected to a ToR switch are in 10.1.0.0/26 subnet. A typical configuration of BGP dynamic neighbors on a ToR will look as follows:

```
neighbor servers peer-group
neighbor servers remote-as 65530
bgp listen range 10.1.0.0/26 peer-group servers
```

At this point, the BGP daemon will begin listening passively on port 179 (the well-known BGP port). If it receives a connection from anyone in the 10.1.0.0/26 subnet that says it's ASN is 65530, the BGP daemon will accept the connection request, and a new BGP session is established.

On the server side, the switch's peering IP address is typically that of the default gateway. For the subnet 10.1.0.0/26, the gateway address is typically 10.1.0.1. Thus, the BGP configuration on the server can be as follows:

```
neighbor ISL peer-group
neighbor ISL remote-as external
neighbor 10.1.0.1 peer-group ISL
```

At this point, the BGP daemon running on the server will initiate a connection to the switch, and as soon as the connection is established, the rest of the BGP state machine proceeds as usual.

Unfortunately, the dynamic neighbors features is not currently supported over an interface; that is, you cannot say `bgp listen interface vlan10 peer-group servers`. Nor is it possible to use the

interface name on the server end, because the trick of using interface names (described in Chapter 3) works only with /30 or /31 subnet addresses, whereas what's used here is a /26 address.

You can limit the number of peers that the dynamic neighbor model supports via the command `neighbor listen limit` *`limit-number`*. For example, by configuring `bgp listen limit 20`, you allow only 20 dynamic neighbors to be established at any given time.

The primary advantage of this model is that it works well with single-attached servers, and when the servers are booted through the Preboot Execution Environment (PXE). Figure 6-1 presents this model.

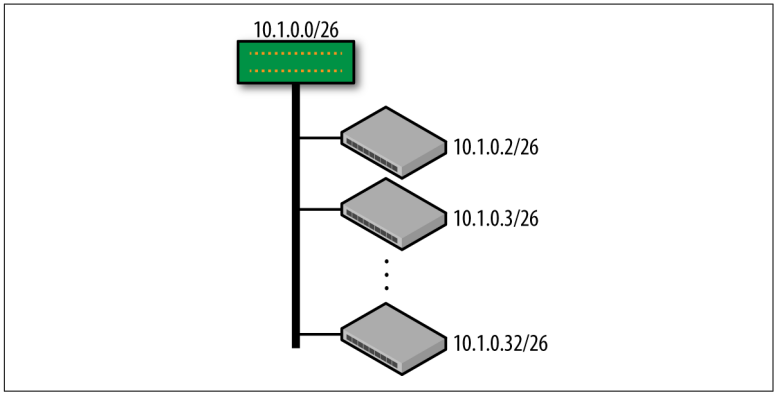


Figure 6-1. BGP Dynamic neighbor over a shared subnet

BGP unnumbered model

Much like BGP session establishment between routers, a BGP session can be established between a server and a switch using BGP unnumbered. Recall from Chapter 4 that BGP unnumbered works in the FRRouting suite without requiring any modification in the Linux kernel.

The model for configuration with BGP unnumbered, shown in Figure 6-2, looks different from the dynamic neighbor version.

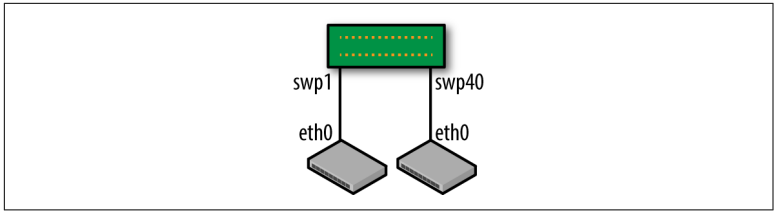


Figure 6-2. BGP unnumbered model of peering with hosts

Unlike the shared subnet model of dynamic neighbors, the BGP unnumbered model has no shared subnet. Just like a router, the server's IP address is independent of the interface and typically assigned to the loopback address. Every server can be assigned an independent /32 address. Because the IPv6 link local address (LLA) is used to peer with the router, there is no need for a shared subnet.

The configuration on the switch side will look something as follows:

```
neighbor peer-group servers
neighbor servers remote-as external
neighbor swp1 peer-group servers
neighbor swp2 peer-group servers
...
```

And the configuration on the server side looks similar:

```
neighbor eth0 remote-as external
```

The main advantage of this approach is that you can build a pure routed data center, with bridging completely eliminated. This model also supports dual-attached servers, with no need to run any proprietary multinode LACP. The main disadvantage of this approach is that DHCPv4 or PXE-booted servers are difficult to support because there is no routing stack during PXE-boot, but the switch doesn't know how to forward packets to a specific server. There are possible solutions, but the explanation is beyond the scope of the book.

The BGP unnumbered model over a shared interface is theoretically possible when the shared link is between a switch and group of servers, but is currently unimplemented.

Routing Software for Hosts

If you're well-versed in network design, you will recognize that in reality, the BGP running on the server really needs to be just a BGP speaker, and doesn't have to implement a full routing protocol with

best-path computation, programming routes into the routing table, and so on. Web-scale pioneers recognized this and ran software such as ExaBGP, which only functioned as BGP speaker, for a long time.

Today more full-featured open source routing suites such as FRRouting and BIRD routing are available for use on Linux and BSD servers. FRRouting supports both BGP unnumbered and dynamic neighbors. The examples used in this chapter relied on FRRouting.

Summary

This chapter showed how we can extend the use of BGP all the way to the hosts. With the advent of powerful, full-featured routing suites such as FRRouting, it is possible to configure BGP simply by using BGP unnumbered, making it trivial to automate BGP configuration across all servers. If you cannot live with the current limitations of BGP unnumbered or you prefer a more traditional BGP peering, BGP dynamic neighbors is an alternative solution. Further, we showed how we could limit any damage that can be caused by servers advertising incorrect routes into the network, advertently or inadvertently.

About the Author

Dinesh G. Dutt is the Chief Scientist at Cumulus Networks. He has been in the networking industry for the past 20 years—most of it at Cisco Systems, where he was a Fellow. He has been involved in enterprise and data center networking technologies, including the design of many of the ASICs that powered Cisco's mega-switches such as Cat6K and the Nexus family of switches. He also has experience in storage networking from his days at Andiamo Systems and in the design of FCoE. He is a coauthor of TRILL and VxLAN, and has filed for over 40 patents.