

O'REILLY®

Second
Edition



Arista Warrior

A Real-World Guide to Understanding
Arista Products and EOS

Gary A. Donahue

1. Preface

- a. Who Should Read This Book
- b. Conventions Used in This Book
- c. Using Code Examples
- d. O'Reilly Online Learning
- e. How to Contact Us
- f. What's Changed
- g. A Quick Note About Versions
- h. A Quick Note About Hardware
- i. A Quick Note About Code Examples
- j. Acknowledgments
- k. Content Disclaimer

2. 1. Why Arista?

- a. A Brief History of Arista
 - i. Key Players
- b. The Needs of a Data Center
- c. Data Center Networking
- d. The Case for Low Latency
- e. What About the Enterprise?
- f. Network-Based Storage
- g. Automation

h. Arista Delivers

 i. Hardware

 ii. EOS

 iii. Bash

 iv. SysDB

 i. Automation

 j. CloudVision

 k. Cognitive Campus

 l. My Personal Take

3. 2. Network Designs

 a. Bow Tie MLAG

 b. Leaf-Spine

 c. Spline

 d. VXLAN/Overlay

 e. Universal Spine

 i. AnyCloud

 f. Conclusion

4. 3. Buffers

 a. Conclusion

5. 4. Merchant Silicon

 a. The Debate

 b. Arista and Merchant Silicon

c. Arista Product ASICs

d. Conclusion

6. 5. Fabric Speed

a. Conclusion

7. 6. Arista Products

a. Switches

i. Switch Names

ii. Power

iii. Airflow

iv. USB

v. Rails

vi. High-Speed Ethernet

vii. Nonblocking Architecture

viii. FlexRoute

ix. AlgoMatch

x. Optics

b. Software Products

i. EOS

ii. vEOS

iii. AnyCloud

iv. cEOS

v. CloudVision

- c. Conclusion
- 8. 7. SysDB
 - a. Conclusion
- 9. 8. Introduction to EOS
 - a. SysDB
 - b. Using EOS
 - c. Conclusion
- 10. 9. Configuration Management
 - a. Configure Replace
 - b. Configuration Checkpoints
 - c. Configuration Sessions
 - d. Conclusion
- 11. 10. Upgrading EOS
 - a. Conclusion
- 12. 11. Link Layer Discovery Protocol
 - a. Conclusion
- 13. 12. Bash
 - a. Some Quick EOS Bash Tips
 - b. Conclusion
- 14. 13. Zero-Touch Provisioning
 - a. ZTP Requirements
 - b. Cancelling ZTP

- c. Disabling ZTP
 - d. Booting with ZTP
 - e. Conclusion
15. 14. Aboot
- a. Password Recovery
 - b. Conclusion
16. 15. CloudVision
- a. The CloudVision Family
 - i. CloudVision eXchange
 - ii. CloudVision Portal
 - iii. Quick Things to Know
 - b. CloudVision Portal
 - i. What CVP Is
 - ii. What CVP Isn't
 - iii. How CVP Works
 - iv. Login Page
 - v. Devices
 - vi. Network Provisioning
 - vii. Events
 - viii. Cloud Tracer
 - ix. Topology
 - c. Conclusion

17. 16. The EOS Extension System

a. Conclusion

18. 17. Multiple Spanning Tree Protocol

a. MST

i. MST Terminology

b. Pruning VLANs with MST

c. Conclusion

19. 18. MLAG

a. MLAG Overview

b. Configuring MLAG

c. Managing MLAG

d. MLAG Consistency

e. MLAG Failover

i. Dual-Primary Detection

f. Bow Tie MLAG

g. MLAG In-Service Software Upgrade

i. Layer 3 with MLAG

h. Spanning Tree and MLAG

i. Conclusion

20. 19. First-Hop Redundancy

a. VRP

i. Basic Configuration

ii. Miscellaneous VRRP Stuff

b. VARP

i. Configuring VARP

c. Conclusion

21. 20. FlexRoute

a. How FlexRoute Works

b. Simulating 800,000 Routes

c. Configuring and Using FlexRoute

d. Conclusion

22. 21. VXLAN

a. Understanding VXLAN

b. Configuring VXLAN

i. VXLAN with Manual Control-Plane

c. VXLAN with CVX

d. VXLAN with EVPN

e. VXLAN with MLAG

f. Conclusion

23. 22. Email

a. Conclusion

24. 23. LANZ

a. Microbursts Visualized

b. Queue Thresholds

c. Conclusion

25. 24. Scheduler

a. Conclusion

26. 25. tcpdump and Advanced Mirroring

a. tcpdump in Linux

b. tcpdump in EOS

c. Advanced Mirroring

d. Filtering Advanced Mirroring Sessions

e. Truncation with Advanced Mirroring

f. Conclusion

27. 26. Tap Aggregation

a. Tap Aggregation from the Command-Line Interface

b. The TapAgg GUI

c. Conclusion

28. 27. Event Manager

a. Description

i. on-boot

ii. on-counters

iii. on-intf

iv. on-logging

v. on-maintenance

vi. on-startup-config

- b. Configuring Event Handlers
- c. Showing the Event Handler Status
- d. Conclusion

29. 28. Event Monitor

a. Using Event Monitor

- i. A Note About Versions
- ii. ARP
- iii. MAC
- iv. Route
- v. LACP
- vi. IGMP Snooping
- vii. MRoute

b. Configuring Event Monitor

c. Conclusion

30. 29. Troubleshooting

a. Logs

b. Performance Monitoring

- i. Tracing Agents (Debugging)
- ii. Turn It Off!

c. CLI Standalone Mode

- i. Arista Support

d. Conclusion

31. 30. eAPI

- a. GAD's Rant About the Fear of Scripting
- b. Expect Scripting
- c. Screen Scraping
- d. Python Data Type Primer
- e. Configuring eAPI
- f. eAPI Web Interface
- g. Scripting with eAPI
- h. Conclusion

32. 31. Containers

- a. Why EOS Containers?
- b. cEOS—EOS in a Container
 - i. Some Things to Watch Out For
- c. Containers in EOS
- d. Conclusion

33. 32. vEOS

- a. VEOS in VirtualBox
 - i. Creating the Base VM
 - ii. Making a Real Lab Through Cloning
 - iii. Building the Interswitch Networks
 - iv. Automating VirtualBox Builds

b. vEOS in an EOS VM

c. Conclusion

34. 33. Aristacisms

a. Arista-Specific Configuration Items

i. There Is No Duplex Statement in EOS

ii. Watch Out for Those Comments!

iii. Some Routing Protocols Are Shut Down by Default

iv. Trunk Groups

b. Virtual Routing and Forwarding

c. Open Flow and Direct Flow

d. Macro-Segmentation Service

e. And, Finally...

f. Conclusion

35. Index

Praise for Arista Warrior

Arista Warrior is a practical approach to networking, providing an overview of Arista hardware platforms and EOS software used to build modern, large-scale networks. The author has created a book which can be used as an educational primer and a day-to-day reference.

The reader will learn about advances in hardware and practical experiences in configuration management and programmability. This book is a must-have for any engineers interested in building software defined networks and I am happy to recommend it.

—Edet Nkposong, Network Engineer

Some time ago, I was lucky to have a chance to work with Arista technologies in their earlier days. Back then, we had nothing at hand but the command reference. A friend passed me the first edition of the Arista Warrior, and since then it hasn't left my desk. Having worked within networking for over 20 years, I got used to the literary and dry style of many books in the field. This one is not just another title on data center network technologies.

Concise and pragmatic, it starts with an overview of switching technologies fundamentals and then explores the flexibility and programmability of the Arista EOS environment. Besides the EOS, the book goes over Arista's hardware platforms at a high level of detail. As an end-to-end guide, it gives readers the essential set of skills to build, automate, and operate the data center networks. After all the years of experience, and using the book myself, I would recommend it to any fellow network engineer!

—Petr Lapukhov, Network Engineer

Arista Warrior

SECOND EDITION

A Real-World Guide to Understanding Arista
Products with a Focus on EOS

Gary A. Donahue



Arista Warrior

by Gary A. Donahue

Copyright © 2019 Gary Donahue. All rights reserved.

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>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Acquisitions Editor: John Devins

Development Editor: Angela Rufino

Production Editor: Christopher Faucher

Copyeditor: Octal Publishing, LLC

Proofreader: Kim Wimpsett

Indexer: WordCo Indexing Services, Inc.

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

July 2019: Second Edition

Revision History for the Second Edition

- 2019-06-28: First Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781491953044> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Arista Warrior*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author, and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-95304-4

[LSI]

Dedication

I dedicate this book to myself because, dammit, I deserve it.

Preface

The examples used in this book are taken from my own experiences as well as from the experiences of those with or for whom I have had the pleasure of working. Of course, for obvious legal and honorable reasons, the exact details and any information that might reveal the identities of the other parties involved have been changed. Because they're guilty as hell. Every last one of 'em.

Who Should Read This Book

This book is not an Arista manual. I will not go into the details of every permutation of every command, nor will I go into painful detail of default timers or counters or priorities or any of that boring stuff. The purpose of this book is to get you up and running with an Arista switch, or even a data center full of them. What's more, this book aims to explain Arista-specific features in great detail; however, it might not go into such detail on other topics such as explaining VLANs, routers, and how to configure NTP, because I've covered those topics at length in *Network Warrior*. I will go into detail if a topic is being introduced here that wasn't covered in *Network Warrior*, such as Multiple Spanning Tree (MST) or Virtual Router Redundancy Protocol (VRRP). Where possible, I have concentrated on what makes Arista switches great. In short, if you want to learn about networking, pick up *Network Warrior*. If you want to know why Arista is stealing market share from all of the other networking equipment vendors (and continues to do so

six years after the first edition was published), buy this book.

This book is intended for use by anyone familiar with networking from any vendor's environment, but most likely from one with that industry-standard command-line interface (CLI) we all know, who is interested in learning more about Arista switches. Anyone with entry-level networking certification (or greater) knowledge should benefit from this book, but the person who will get the most from this book is the entrenched admin, engineer, or architect who has been tasked with building an Arista network. My goal in writing *Arista Warrior* is to explain complex ideas in an easy-to-understand manner. I've taught many classes on Arista switches, and I see trepidation and fear of the unknown in students when the class begins. By the end of the class, I have students asking when the next class will be available and if I can get them Arista T-shirts. In the first printing of this book, I commented that I couldn't get shirts. Now that I work for Arista, I can tell you that if you take one of my classes, you might get a T-shirt! I hope you will find this book similarly informative, lack of free vendor swag notwithstanding.

As I wrote in *Network Warrior*, I have noticed over the years that people in the computer, networking, and telecom industries are often misinformed about the basics of these disciplines. I believe that in many cases, this is the result of poor teaching or the use of reference material that does not convey complex concepts well. With this book, I hope to show people how easy some of these concepts are. Of course, as I like to say, "It's easy when you know how," so I have tried very hard to help anyone who picks up my book understand the ideas contained herein.

Let's be brutally honest; most technology books suck. What drew me to O'Reilly in the first place is that most of theirs don't. From the feedback I've received over the years since first writing *Network Warrior*, it has become clear to me that many of my readers agree. I hope that this book is as easy to read as my previous works, and at least as entertaining. For those of you who have met me during one my classes, I apologize in advance for the fact that you'll hear my voice in your head as you read. As someone who's lived his entire life with my own voice in my head, I know that nobody wants that.

My goal, as always, is to make your job easier. Where applicable, I will share details of how I've made horrible mistakes in order to help you avoid them. Sure, I could pretend that I've never made any such mistakes, but anyone who knows me will happily tell you how untrue that would be. Besides, stories make technical books more fun, so dig in, read on, and enjoy watching me fail, because when I do, I tend to fail in a big way.

This book is similar in style to *Network Warrior*, with the obvious exception that there is no (well, very little, really) Cisco content. In some cases, I include examples that might seem excessive, such as showing the output from a command's help option. My assumption is that people don't have Arista switches sitting around that they can play with. This is a bit different than all of the other entrenched vendors' worlds, where you can pick up an old switch on the internet for little money. Arista is still a relatively new company (12 years old as of this writing), and finding used Arista switches will probably be tough. Hopefully, by including more of what you'd see in an actual Arista switch, this book will help those curious about them.

NOTE

One of the great changes that Arista made from the days of the first edition of Arista Warrior is that you can now get a copy of vEOS or cEOS simply by registering a guest account on the Arista website. This is a positively huge development for those of us who want to get certified or just want to learn more about Extensible Operating System (EOS).

Lastly, I'd like to explain why I wrote this book. When I wrote the first edition of this book, I didn't work for Arista, I didn't sell Arista gear, and Arista had not paid me to write this book. I now work for Arista as an instructor and course developer. Arista did not dictate the contents of this book, and I even went so far as to insist that I continue to have some autonomy in my writing career when hired. If I see something that I don't like, I'll write about it, but to be completely up front, because I work for Arista, I am likely to present the issue to them for correction before publishing anything negative. The only restriction I have is that I cannot post any trade or company secrets. Telling you things like, "Mark Berly is a robot sent from the future to destroy us all," would be similarly unethical, and I just won't do it. Much.

NOTE

For those not in on the joke, Mark Berly is one of the very early employees at Arista and was the one who recommended me for the job. He's an all-around great guy and is part of what makes Arista great. He also said, "Holy crap, you got old!" the last time I saw him, so he deserves a light-hearted jab. In print. Forever.

Some time ago, before my tenure at Arista began, a client had me do a sort of bake-off between major networking equipment vendors (Mark

Berly was my Arista systems engineer). We brought in all the big names, all of whom said something to the effect of, “We’re usually up against Arista in this space!” Because every one of the other vendors inadvertently recommended Arista, we contacted the company, got some test gear, and went out to visit its California office.

I’ve been in IT for more than 30 years, and I’ve been doing networking for more than 25. I’m jaded, I’m grouchy, and I distrust everything I read. I’ve seen countless new ideas reveal themselves as a simple rehashing of something we did with mainframes. I’ve seen countless IT companies come and go, and I’ve been disappointed by more pieces of crappy hardware with crappy operating systems than most people can name. I’ve been given job offers by the biggest names in the business and turned them all down. Why? Because big names mean nothing to me aside from the possibility of another bullet added to my résumé.

Nothing impresses me, nothing surprises me, and nothing gets past me. But when I walked out of Arista after three days of meeting with everyone from the guys who write the code to the CEO and founders themselves, I was impressed. Not only impressed, but excited! I’m not easily sold, but I walked out of there a believer, and in the eight or so years since that first introduction (holy crap—I *am* old!), nothing has caused me to change my perception of Arista and its excellent equipment.

When I began writing, there were no Arista books out there. I felt that I could write one that people would enjoy while doing justice to the Arista way of doing things. As you read this book, I hope that you’ll get a feel for what that way is. So many years later, there really aren’t

any other Arista books out there, though I expect that might soon change. To all of the up-and-coming authors writing about Arista, I salute you with a hearty, “Neener, neener, I was here first!”

Though I’m obviously a fan, these devices are not perfect. I’ll show you where I’ve found issues and where there might be *gotchas*. That’s the benefit of me not being paid by Arista (at least not when it comes to this book)—I tell it like it is. To be honest, though, in my experience, Arista would tell you the very same things, which is one of the first things that impressed me about the company. That’s why I wrote this book. It’s easy for me to write when I believe in the subject matter. As anyone who knows me will attest, it’s impossible for me to evangelize something that I don’t believe in.

Enough blather—let’s get to it!

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Used for new terms where they are defined, for emphasis, and for URLs

Constant width

Used for commands, output from devices as it is seen on the screen, and samples of Request for Comments (RFC) documents reproduced in the text

Constant width italic

Used to indicate arguments within commands for which you should

supply values

Constant width bold

Used for commands to be entered by the user and to highlight sections of output from a device that have been referenced in the text or are significant in some way

NOTE

This element signifies a general note.

WARNING

This element indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “*Arista Warrior, 2e* by Gary A. Donahue. Copyright 2019 Gary Donahue, 978-1-491-95304-4.”

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

O'Reilly Online Learning

NOTE

For almost 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, conferences, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, please visit <http://oreilly.com>.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at
<http://bit.ly/arista-warrior-2e>.

To comment or ask technical questions about this book, send email to
bookquestions@oreilly.com.

For more information about our books, courses, conferences, and news, see our website at <http://www.oreilly.com>.

Find us on Facebook: <http://facebook.com/oreilly>

Follow us on Twitter: <http://twitter.com/oreillymedia>

Watch us on YouTube: <http://www.youtube.com/oreillymedia>

What's Changed

Updating this book was a challenge for a variety of reasons, not the least of which being that there is a lot of new material to cover. There's new hardware, new versions of EOS, new features, and new software products, all of which should really be covered. Alas, there is only so much time and so much space, and so not everything I would like to write about could be included.

My primary goals were as follows:

- Update all of the drawings to use Arista icons and not those awful things that I made on my own last time
- Simplify and reorganize many drawings
- Update all of the code examples to a modern revision of code
- Update all the examples to use 7280R switches
- Reformat the hardware chapter to talk about features and not specific models

Some chapters from the first edition have been removed due to either the topics being molded into other chapters, the content being dated, or there simply not being space for a topic that I considered cut-worthy.

Chapters that have been removed include the following:

- Python (merged and replaced with eAPI)
- Routing (generic routing removed—the rest rewritten to become FlexRoute)
- Sflow
- XMPP (called CloudVision in the first edition)
- QoS

- VM Tracer
- Access-Lists

Some of these chapters, like Access-Lists, Sflow, and QoS, don't really add anything to the book because they're pretty generic and easily learned from the manual. I also feel that they don't really contribute to how Arista is different, so I removed them in favor of the new chapters. If you'd like to read about these topics, the first edition of Arista Warrior is always available as soft copy.

I've added the following chapters (in no particular order):

- Config Management
- eAPI
- CloudVision
- VXLAN
- FlexRoute
- TapAgg
- Advanced Mirroring
- vEOS
- Containers
- Network Design

Paul Valery said, “A poem is never finished, only abandoned.” That’s exactly how I feel about this book. If I could add everything I wanted, the book would be 2,800 pages and cost a small fortune. It would also never be done, and it’s already far overdue, so at some point I had to

stand up and yell “enough!” much to the amusement of everyone in Starbucks at the time.

A Quick Note About Versions

When I began writing the first edition, EOS version 4.8.3 was the state-of-the-art release from Arista. As I continued writing over the course of about a year, new versions of code came out that made it tough to keep up, but the final book had mostly EOS 4.9 examples. When I began the second edition of Arista Warrior, I upgraded all code examples to 4.16.6M, the most current public release available at the time. I then had a solid year (or two) of extensive travel and overwhelming personal...stuff...that caused a delay in the finishing of this tome, after which I then had to update all the examples again, this time to a minimum of 4.19.5M, which then became 4.21.1F (with some odd 4.20 here and there).

A Quick Note About Hardware

In the first edition of this book I covered specific hardware models. I have not done that in this edition because there are now too many devices and products to cover in a book of this size and it would quickly become obsolete given the pace at which Arista comes out with killer new products.

All of the examples used in this book that are switch based are shown on fixed-configuration Arista devices. This is for a couple of reasons, the biggest having to do with the fact that I have practically unlimited access to 7280R switches and little easy access to chassis switches. As

such, there is a lack of coverage of topics having to do with line cards and redundant supervisors. Line cards are covered here and there, but only as needed in a given topic where the behavior changes in a chassis. An example would be Tap Aggregation wherein a chassis can be configured in hybrid mode as opposed to a fixed-configuration switch that can only be configured in exclusive mode.

Really, this is a book about EOS and its many features, but some of those features are based on the availability of a feature in a particular Application Specific Integrated Circuit (ASIC), which will be noted where applicable.

A Quick Note About Code Examples

In many of the examples involving code, I've had to slightly alter the output in order to make it fit within the margins of this book. I've taken great pains to not alter the meaning of the output, but rather to alter only the format. For example, in the output of `show top`, the output includes lines that say something to the effect of:

```
last five minutes: 18.1%, last five seconds 3.1%.
```

To make the example fit, I might alter this to read:

```
last five mis: 18%, last five secs 3%.
```

Sometimes, there are no easy spaces to remove, so I'll replace easy-to-understand sections with ellipsis in square brackets. For example, this output line (from the CLI `dir` command) is so long that the second line wraps and there are few spaces, so the natural option looks like this:

```
-rwxrwx--- 1 root eosadmin 3267 Jan 15 20:45 /mnt/flash/startup-config
-rwxrwx--- 1 root eosadmin 3267 Jan 15 20:45
/mnt/flash/startup-config_2019-01-15_20-45-49
```

I will change the output to look something like this so that the output remains on one line:

```
-rwxrwx--- 1 root eosadmin 3267 Jan 15 20:45 /mnt/flash/startup-config
-rwxrwx--- 1 root eosadmin 3267 Jan 15 20:45 /mnt/flash/sta[...]2019-01-
15_20-45-49
```

Any changes I've made will in no way alter the point of the output, but the output might look slightly different than what you might see on your screen if you run the same command. In some cases, such as the output of `tcpdump`, I've simply changed the position in which the line wraps from, say, 80 columns to 70. Again, this should only have the effect of possibly making the output look different than what you would see when using a terminal emulator without such restrictions.

Acknowledgments

Writing a book is hard work—far more difficult than I ever imagined. Though I spent countless hours alone in front of a keyboard, I could not have accomplished the task without the help of many others.

I would like to thank my lovely wife, Lauren, for being patient, loving, and supportive. Thank you for helping me achieve another goal in my life, which as anyone who's married knows is code for “putting up with my crap.”

In the first edition, I thanked my daughters, Meghan and Colleen, for trying to understand that when I was writing, I couldn't play video

games, go geocaching, or do other fun things. These days, hauling my nerdy ass all around the globe spreading the gospel (that I wrote) often makes it seem like I'm rarely home. But then, they're now both teenage girls (one is in college, and the other is on the way!), and who wants to be home for that? Seriously, though, I hope you both appreciate that I've always endeavored to be there for your important events, and when I couldn't be, you know that I was there in spirit, and by "in spirit," I mean "with money." The fact that money is available either directly or indirectly as a result of my writing means I can legitimately call myself a writer while providing debt-free college for you both, and few things in my life have been more rewarding. May you both be able to follow your dreams and change the world.

I would like to thank my mother because she's my mom and because she never gave up on me, always believed in me, and always helped me even when she shouldn't have. We miss you.

I would like to thank my father for being tough on me when he needed to be, for teaching me how to think logically, and for making me appreciate the beauty in the details. I have fond memories of the two of us sitting in front of my RadioShack Model III computer while we entered basic programs from a magazine. I am where I am today largely because of your influence, direction, and teachings. You made me the man I am today. Thank you, Papa. I miss you.

This second edition would not have been possible without the significant help from the following people at Arista Networks: Mark Berly, Darrin Thomason, Adam Levin, Rich Parkin, and Anshul Sadana. Of special importance to the continued success of this book is

the ongoing support of Katie Smith and Brooke Decker because they rock, and that's all you need to know. A quick shout-out to Namita Kallianpurkar, Lynne Ealden, Rio Ocampo, Samantha Mendoza, Daksha Rajagopalan, and Megan Torrcampo, who didn't have anything to do with this book directly so much as they made my daily life on the road substantially more pleasant by continuing to make sure everything worked as it should. Without me having to worry about that stuff, I arguably had more time to write. I say arguably because I still missed too many deadlines to count, but that's on me.

I'd also like to thank the many developers who have answered my many (often strange) questions. On the top of that long list has got to be Andre Pech, who constantly surprises me by continuing to answer my queries even though he's one of the first 10 Arista employees and definitely has better things to do with his time. It's people like Andre that continue to make Arista great.

Special thanks to Fred Hsu who manages to get me excited all over again every time he describes some new feature or capability. It is through his excitement and explanations that the Container chapter was born.

Rich Parkin gets a special thanks for showing me the cool IP range application capability that I had somehow missed for years. He also had a significant impact in helping me with CloudVision.

Corey Hines also get special mention for helping me with CloudVision. As lead of the CloudVision working group, there are few people at Arista more qualified on that product.

Thanks to Sarthak Shetty and Kelsey Skemp from Arista TAC for helping me out with a particularly vexing issue that I would have never resolved on my own. Arista TAC is easily the best support group I've ever worked with, and I say that as someone who used to be a TAC engineer back in the days when mainframes walked the earth.

Terren Sapp gets a nod for helping to get some virtual environments up to snuff in time for me to actually make a deadline. I'm sure my editor would like to send you a cake for making that rare milestone possible.

Liz Stein gets a nod for making me laugh with some of her tech-editor comments. And, you know, for supplying useful tech-edits.

Josh Frank gets a special mention because although he had time to edit only one chapter, he managed to comment on something that no other editor had found that had been wrongly printed in the first edition!

I'd also like to personally thank Jayshree Ullal, CEO of Arista, for introducing me to an endless stream of customers as "The Arista Warrior." If I could get it past the NSA, I'd walk around the office in California with armor and a sword, but I have enough trouble getting detained by various travel-related policing agencies.

A special word of thanks is needed for Mark Berly. I met with Mark many times when writing the first edition, and I probably emailed him 30 times a day for six months. It takes a special kind of person to tolerate me in the first place, but putting up with my nonstop questions takes either someone who is as nuts as I am or someone who really loves the subject at hand, or both. Thank you for taking the time to

answer my many hundreds of questions. This book would have sucked without your many helpful insights. Even though you really had nothing to do with the second edition, there wouldn't be a second edition without the first, so thanks.

I would like to thank all the wonderful people at O'Reilly. Writing this book was a great experience, due in large part to the people I worked with there. This is my fourth project with O'Reilly, and it just never stops being great.

I would like to once again thank my good friend, John Tocado, who hopefully by now already knows why. Thank you.

Thanks to Lois (the boss, who on her first day as a barista was warned about me by my own wife), Emmy, Kali (who says, “hi!” every day and makes killer foam), Lindsey, Garrett, Justin, Steven, Nic, Kiersten, Lili, Nick, Mikey, Josie (the only person to ever hand-deliver a drink to my table), Bridget, Jade (who told me my name was correct, which made me laugh), Jessye, Hannah (who makes a great cappuccino), Stephanie, Cruz, and Ryan, all of the Chester Starbucks where I spent far too much time working and writing. They all knew I was coming for my coveted seat in the corner when they saw my mobile order of a short cappuccino and a Cheese and Fruit Protein Box come in. Sadly, they have repeatedly denied my request to have that table cordoned off with velvet rope.

I still wish to thank everyone else who has given me encouragement. Living and working with a writer must, at times, be maddening. Under the burden of deadlines, I've no doubt been cranky, annoying, and

frustrating, for which I apologize.

My main drive for the past few months has been the completion of this book. All other responsibilities, with the exception of health, family, and work, took a backseat to my goal. Realizing this book's publication is a dream come true for me. You might have dreams yourself, for which I can offer only this one bit of advice: work toward your goals and you will realize them. It really is that simple.

Remember the tree, for the mighty oak is simply a nut that stood its ground.

Finally, if you enjoy my writing and would like to read more, please check out my various blogs:

- Guild Guitars
- Cozy Tales
- Tales of a Butter-Licking Ferret-Dog
- And, finally, GAD's Ramblings, which is a derivation or source for all of the above

Content Disclaimer

Any opinions stated in this book are 100% my own and do not reflect that of Arista or any other person or entity. I am an employee of Arista as of this publishing, but Arista has not told me what I can or cannot write. The content was edited by Arista employees, but for technical accuracy and to ensure that no trade secrets have been revealed. Even grammar was outside the purview of the tech editors, so if anything is

misspelled or I end up talking in colloquialisms, that's entirely my doing. The copy editors at O'Reilly are pretty great, though, so hopefully that won't happen (much) either.

Simply put, this is a book *about* Arista, not a book *by* Arista. That's an important distinction to me, and I believe it's important to Arista because they have encouraged me to write without telling me what to write and, perhaps more important yet, without telling me what not to write.

Chapter 1. Why Arista?

If you're reading this book, you have an interest in Arista products for any number of reasons. My goal is for you to understand why Arista is here, why it should be taken seriously, and why its products are selling like crazy. So let's get started by explaining how it all began.

A Brief History of Arista

Arista Networks is a successful networking equipment company that's been around only since 2004. It takes something special to succeed in an industry dominated by well-entrenched companies, many of which have been on top for decades. Certainly, a good product is needed, but that product and everything it takes to produce it comes from people. The people are what make Arista great.

Key Players

Please indulge me while I give you a quick tour of some of the key players at Arista, because having met many of them, I firmly believe that these people infect everyone around them with the same attitudes, excitement, and belief in what they're doing. There are three people responsible for the creation of Arista Networks: Andy Bechtolsheim, David Cheriton, and Ken Duda.

ANDY BECHTOLSHEIM

Andy Bechtolsheim cofounded a company called Sun Microsystems in 1982. You might have heard of it. In 1995, he left Sun to found a company called Granite Systems. This new company made its mark by developing (then) state-of-the-art high-speed network switches. In 1995, Cisco acquired Granite Systems for a cool \$220 million. With the sale, Andy became vice president and general manager of the Gigabit Systems Business Unit, where he stayed until 2003. He left Cisco in December of that year to found Kealia, Inc., with a Stanford professor named David Cheriton. Kealia was later acquired by Sun Microsystems, where Andy returned to the role of senior vice president and chief architect. In 2005, Andy cofounded Arastra, which later changed its name to Arista Networks.

Andy has an MS in computer engineering from Carnegie Mellon University and a PhD from Stanford University.

Andy Bechtolsheim is a multibillionaire Silicon Valley visionary. He has either designed or had a hand in the creation of some of the most significant computing and networking devices of the past 30 years. Andy and David Cheriton were the two initial investors in Google. Each of their \$100,000 investments are now worth...well, let's just say they made their money back and then some. I've had the pleasure of talking with Andy on multiple occasions, and I can tell you that he's one of the smartest and most humble people I've ever met. I also stepped on his foot once before a speaking engagement, so I've got that going for me.

DAVID CHERITON

David Cheriton is a Stanford University computer science professor

who has an amazing knack for spotting and investing in successful startups. David cofounded Granite Systems with Andy Bechtolsheim, and the two have started other successful companies, including the aforementioned Kealia. David served as a technical advisor for Cisco for seven years and was the chief architect for the ASICs used in the Catalyst 4000s and 4500s. He has also served as a technical advisor for companies such as Sun, VMware, and Google. David is one of the original founders of Arastra, later renamed Arista Networks. He served as chief scientist for Arista until the IPO in 2014, at which point he left the company for reasons that are outside the scope of this book.

David has multiple inventions and patents to his name, has a PhD in computer science from the University of Waterloo, and has been at Stanford since 1981.

Given the track record of Andy and David and the fact that these two men funded the new company without any other investors, it would seem that Arista is destined for greatness, but the story doesn't end here.

KEN DUDA

Ken Duda is a founder, chief technology officer, and senior vice president of software engineering at Arista. Prior to founding Arastra (now Arista), Ken was CTO of There.com, where he designed a real-time 3-D distributed system that scaled to thousands of simultaneous users. I have no idea what that means, but it sure sounds cool.

Ken was the first employee of Granite Systems and, while working at Cisco, led the development of the Catalyst 4000 product line.

Ken has three simultaneous engineering degrees from MIT and a PhD in computer science from Stanford University.

Much of what you will read in this book about EOS is a result of Ken Duda's vision. I met Ken while visiting Arista (along with many of the other people mentioned in this chapter), and within minutes, I realized that he was living the dream. Well, to be fair, maybe it was my dream, but what I saw was a seriously smart guy who knew *the right way to do it* and who had the freedom to do just that. I might be a hack writer now, but I went to school for programming (COBOL on punch cards, thank you very much) and loved being a programmer (we weren't called developers back then). I gave up programming because I grew tired of having to fix other people's crappy code. I wanted to write amazing new systems, but companies weren't looking for that—they wanted grunts to fix their crappy code.

Ken not only gets to write the kind of code he likes, but he gets to design an entire networking equipment operating system from the ground up. When I first visited Arista, I drilled him with questions. Wouldn't that delay delivery? Wouldn't investors complain? Didn't you ever get rushed into finishing something early to be first to market? As he answered my questions, it all started to become clear to me. There were no crazy investors demanding artificial deadlines. These guys had decided to do it the right way and not to deviate from that course. I also realized that everyone at Arista felt the same way. It was my meeting with Ken Duda that started the idea in my mind to write this book. Someone had to tell the world that companies like this could thrive, because in my almost 30 years in this industry, I can tell you that Arista is the first company I've seen that *does it the right way*.

JAYSHREE ULLAL

The three founders certainly set the direction for Arista as a whole, but Jayshree keeps the place running. Jayshree Ullal is the president and CEO of Arista Networks. She was senior vice president at Cisco, where she was responsible for Data Center Switching and Services, including the Cisco Nexus 7000, the Catalyst 4500, and the Catalyst 6500 product lines. She was responsible for \$10 billion in revenue, and reported directly to John Chambers, CEO of Cisco.

Jayshree has a BS in electrical engineering from San Francisco State University and an MS in engineering management from Santa Clara University.

Jayshree was named one of the “50 Most Powerful People” in 2005 by *Network World Magazine* and one of the “Top Ten Executives” at VMWorld in 2011. She has garnered many awards, including one of the 20 “Women to Watch in 2001” by *Newsweek* magazine. According to *NewsIndiaTimes*, she is the world’s first female Indian-American billionaire.

I can hear you now saying, “blah blah blah, I could read this on Wikipedia.” But consider this: Arista is a company peopled by mad scientists who just happen to work in legitimate jobs doing good work. Jayshree keeps them all in line and keeps the business not only humming but also prospering. Having managed teams and departments of both developers and engineers, I know what a challenge it can be. She makes it look easy.

All of these people are powerful forces in the networking and IT

worlds, and all of them manage to make time to meet with prospective customers and even speak during classes held onsite at Arista. I've been in both situations and have seen this for myself.

You can read more about Arista and the management team at [Arista's website](#).

The Needs of a Data Center

So, what's the big deal about data centers? Why do they need special switches anyway? Can't we just use the same switches we use in the office? Hell, can't we just go to Staples and buy some switches from Linksys, Netgear, or D-Link or something?

Believe it or not, I've had this very conversation on more than one occasion with executives looking to save some money on their data center builds. While it might be obvious to me, I quickly learned that it's not apparent to everyone why data centers are unique.

Data centers are usually designed for critical systems that require high availability. That means redundant power, efficient cooling, secure access, and a pile of other things. But most of all, it means no single points of failure.

Every device in a data center should have dual power supplies, and each one of those power supplies should be fed from discrete power feeds. All devices in a data center should have front-to-back airflow, or ideally, airflow that can be configured front to back or back to front. All devices in a data center should support the means to upgrade,

replace, or shut down any single chassis at any time without interruption to the often-extreme Service-Level Agreements (SLAs). In-Service Software Upgrades (ISSU) should also be available, but this can be circumvented by properly distributing load to allow meeting the prior requirement. Data center devices should offer robust hardware, even Network Equipment Building System (NEBS) compliance where required, and robust software to match.

Although data center switches should be able to deliver all of those features, they should also not be loaded down with features that are not desired in the data center. Examples of superfluous features might include Power Over Ethernet (PoE), backplane stacking, VoIP gateway features, wireless LAN controller functions, and other generally office-specific features.

NOTE

Note that this last paragraph greatly depends on what's being housed in the data center. If the data center is designed to house all the IT equipment for a large office, then PoE and WAN Controllers might be desirable. Really, though, in a proper data center, those functions should be housed in proper dual-power-supply devices dedicated to the desired tasks.

Even though stacked switches seem like a great way to lower management points and increase port density, you might find that switches that support such features often don't have the fabric speed or feature set to adequately support a data center environment. I've made a lot of money swapping out closet switches for data center-class switches in data centers. Data centers are always more resilient when

using real data center equipment. If you don't pay to put them in from the start, you'll pay even more to swap them in later.

Data Center Networking

VMware really shook up the data center world with the introduction of vMotion. With vMotion, virtual machines (VMs) can be migrated from one physical box to another, without changing IP addresses and without bringing the server offline. I have to admit, that's pretty cool.

The problem is that in order to accomplish this, the source and destination servers must reside in the same VLANs. That usually means having VLANs spanning across physical locations, which is just about the polar opposite of what we've spent the past 20 years trying to move away from!

Data center switches should support, or have the ability to support, at least a subset of data center-specific technologies. If your executive comes in and says that you need to support some new whizbang data center technology because he read about it in *CIO magazine* on the john that morning, having a data center full of closet switches will mean a rough conversation about how he bought the wrong gear.

The Case for Low Latency

Low latency might seem like a solution in need of a problem if you're used to dealing with email and web servers, but in some fields, microseconds mean millions: millions of dollars, that is.

I talk about trading floors later on in this book, and some of Arista's biggest customers use Arista switches in order to execute trades faster than their competitors. But think about other environments in which microseconds translate into tangible benefits—environments such as computer animation studios that can spend 80 to 90 hours rendering a single frame for a blockbuster movie, or scientific compute farms that might involve tens of thousands of compute cores. If the network is the bottleneck within those massive computer arrays, the overall performance is affected. And imagine the impact that an oversubscribed network might have on such farms. I've never had the pleasure of working in such environments, but I've taught people who do, and I can tell you that dropping packets is frowned upon.

Sure, those systems require some serious networking, but you might be surprised how much latency can affect more common applications. iSCSI doesn't tolerate dropped packets well, nor does it tolerate a lot of buffering. Heck, even Network Attached Storage (NAS), which can tolerate dropped packets, is often used for systems and applications that do not tolerate latency well. Couple that with the way that most NAS are designed (many hosts to one filer), and things like buffering become a huge issue. Not only have I seen closet switches fail miserably in such environments, I've seen many data center-class switches fail, too.

In 2018, Arista acquired the company Metamako, which, according to the press release, is a leader in low-latency, field-programmable gate array (FPGA)-enabled network solutions. As we approach 2020, Arista still maintains its edge in the industry with respect to low-latency switching.

What About the Enterprise?

Arista is absolutely used in the enterprise, but before about 2018 Arista's focus was mostly in the data center and before that in places like Wall Street. That is rapidly changing with things like the acquisition of Mojo Networks and the introduction of concepts like the Cognitive Campus, which I'll cover in a bit.

Network-Based Storage

NAS was developed in the early 1980s as a means for university students to share porn between systems. OK, I totally made that up, but I'd be willing to bet that it was one of the first widespread uses of the technology. NAS protocols like Network File System (NFS) really were developed in the early 1980s, though, and although they've come a long way, they were not originally designed to be a solution for low-latency, high-throughput storage. Compared with more low-level solutions such as Fibre Channel, NAS tends to be slow and inefficient.

Still, NAS is comparatively inexpensive, it doesn't require special hardware on the server side, and many vendors offer specialized NAS solutions aimed at centralizing storage needs for scores, if not hundreds, of servers. NAS is a reality in the modern data center, and the networks that NAS rides on must be robust, offer low latency, and, whenever possible, not drop packets. Even with nonblocking 10 Gb architectures, it can be easy to oversubscribe the 10 Gbps links to the NAS devices if many servers make simultaneous 10 Gbps reads or writes. Hell, in today's networks, that can even be 100 Gbps, or 400 Gbps!

Automation

As I put the finishing touches on this edition, I find myself in the middle of 2019. For the past six years or so, I've been ranting to anyone who will listen that network engineers need to learn to code or I will script them out of a job. Read [Chapter 30](#) to see why, but while you consider that, think about the fact that in my humble opinion, no other networking vendor supports automation better than Arista does. With an Arista switch out of the box, I can write Python scripts, Bash scripts, eAPI scripts, command-line interface (CLI) scripts (by using the interpreter directive [shebang], `#!/usr/bin/CLI` in Bash), and I can combine them in clever ways if need be. I can automate Arista builds using tools like Ansible, Chef, Puppet, and Arista's own CloudVision.

Arista Delivers

So how does Arista deal with the requirements outlined in this chapter? Here's a short list to whet your appetite. Each one of these topics is covered in detail within this book, so here I'll just supply a list with a brief explanation of each feature and a reference to the chapter in which the topic is covered in more detail.

Hardware

Arista switches all have dual power supplies, hot-swappable and (usually) reversible airflow fans, completely nonblocking fabrics (even the eight- and twelve-slot chassis switches!), and merchant silicon. In almost every case, they are physically smaller, weigh less, consume

less power, and often cost less than comparable switches from other manufacturers; although as you'll come to learn, there really are no other switches that compare, though as the industry has moved to adopt merchant silicon, that is beginning to change. Sure, Arista makes great hardware, but the real difference is in the operating systems. See [Chapter 6](#) for more information.

EOS

The Extensible Operating System (EOS) offers an industry-standard CLI while offering the power, flexibility, and expandability of Linux. Man, what a mouthful of marketing buzzwords that is. Let's cut the BS and tell it like it is: EOS is Linux, with an industry-standard CLI. Actually, even that barely tells the whole story. Arista switches run Linux. They don't run some stripped-down version of Linux that's been altered beyond recognition—they run Linux. Some other vendors say that their operating system (OS) is based on Linux, and I guess it is, but on an Arista switch, you can drop down into the *Bash* shell and kill processes if you're so inclined. Hell, you can even spawn another CLI session from Bash, write scripts that contain CLI commands, send email from CLI, pipe Bash commands through the CLI, and do a host of other exciting things, all because the switch runs Linux and because the developers care about one thing above all else: *doing things the right way*.

Arista hardware is amazing, but EOS makes these devices profoundly different than any other vendor's offerings. See Chapters [8](#), [9](#), [10](#), [16](#), and many others for more information.

Bash

OK, so I blew the surprise with my EOS fan-boy ravings, but yes, you can issue the `bash` command from the CLI and enter the world of Linux. It's not a Linux simulator either—it's Bash, in Fedora Core Linux. You can even execute the `sudo shutdown -r now` command if you want, and you know you want to. All your other favorite Linux commands are there too: `ps`, `top`, `grep`, `more`, `less`, `vi`, `cat`, `tar`, `gunzip`, and `python` just to name a few. But not `perl`—unless you want to add it, in which case you can, because it's Linux.

The fact that these switches run Linux is such a big deal that I recommend learning Linux to my clients when they're considering Arista switches. Of course, the beauty of EOS is that you don't need to know Linux, thanks to the CLI, but trust me when I say you'll be able to get much more out of your Arista switches with some good Linux experience. The best part? Since writing the first edition, I now write training material for Arista, and we will be happy to teach you Linux in a class that's been specially tailored for networking engineers. Actually, I'd probably have Adam Levin teach you that class given that he wrote it and he's just that much better at it than I am. Linux is so important in the world of modern networking that I wouldn't be surprised if Linux knowledge became a requirement in networking certifications moving forward. See [Chapter 9](#) for more information.

SysDB

SysDB is one of the main features that makes EOS and Arista switches great. Simply put, SysDB is a database on the switch that holds all of the critical counters, status, and state information necessary for

processes to run. The processes read and write this information to and/or from SysDB instead of storing it locally. If another process needs the information, it gets it from SysDB. Thus, processes never need to talk to each other; they communicate through SysDB. This dramatically lowers the possibility of one process negatively affecting another. Additionally, if a process dies, it can restart quickly without having to reinitialize all values, because it can read them all from SysDB. See [Chapter 7](#) for more information.

Automation

In my humble opinion (some would say I'm incapable of a humble opinion), Arista leads the industry in automation capabilities. Because EOS is actually Linux, you can use tools like Chef, Puppet, and Ansible. Arista embraces the principles of DevOps and NetOps and has many ways of automating just about anything you can think of in EOS. You'll see examples of me doing this in this book using one of my favorite Arista features, eAPI. You don't need to learn about eAPI to automate EOS, though, because you can write Python and Bash scripts natively in Linux, and because there's a Bash command called `cli` that we cover later on, you can even script CLI commands in those programming languages. If you're a hardcore developer, you can even write your own EOS agents using the EOS SDK. If you want to automate but you don't like to code, you can use CloudVision, as well. See Chapters [15](#), [28](#), and [30](#) for more information.

CloudVision

CloudVision is Arista's software solution for managing and maintaining your network. With its ability to centralize reporting, management, and even telemetry, not to mention doing things like rolling back the entire network configuration (as in hundreds of devices) to a previous point in time, CloudVision is a powerful tool that allows you to automate a litany of tasks without knowing how to code, all through the use of a web-based GUI. Even better, if you do know how to code, you can make CloudVision even more powerful by automating in a more customized fashion. With the addition of features like Bug Alerts, Macro-Segmentation, Topology Views, and a pile of other nice features, CloudVision can be a powerful solution for increased visibility and control of your network. See [Chapter 15](#) for more information.

Cognitive Campus

In 2018, Arista acquired the WiFi company Mojo Networks, Inc., which was a big step toward Arista expanding into the enterprise space. With the addition of *Cognitive Wifi*, PoE switches, and the cloud-controlled infrastructure, Mojo has changed the campus WiFi landscape by moving away from proprietary controllers and protocols.

In addition to the Mojo acquisition, CloudVision allows something Arista calls the *Cognitive Management Plane*, which promises to deliver many of the lessons learned from the data center into enterprise networks.

Although released too late in the publishing process to be included in this book, the *Cognitive Campus* initiative looks like a game-changer

to me. If Arista can change the way that enterprise networks operate the way it did with the data center, and I see no reason why it can't, Arista once again stands to be the leader of that space, too. There are no chapters on this topic yet because, as I just mentioned, it was just coming into existence as this book was being finished.

My Personal Take

Before I worked at Arista, I was an independent consultant who moonlighted as a writer for no other reason than I liked to write. We made the trip out to Arista to get the executive briefing treatment, just like we had with the other big players at the time (around 2011, if I recall). I'd been in the networking world for a long time, and all those big players had pretty much annoyed me with their stories, "secret sauce," and marketing double-speak. Arista was different, though.

When I asked another vendor about how a feature worked, I got a speech about how secret their sauce was (that's not a metaphor—that actually happened). When I asked Ken Duda about how a feature worked, he said, "I didn't write that code, let me get the person who did." That person then came in, opened their laptop, and showed me the code. I was floored.

As someone with a background in programming, Linux, and networking, seeing all of those worlds coming together in a single place got me very excited in a way that the CCIE-level people sitting with me didn't understand. This was game-changing! It was those experiences that made me want to work there and made me want to write the first edition of this book. I was a successful self-employed

consultant. I hadn't wanted to work for anyone but myself for years, maybe even decades; I'd been to Arista's headquarters in California multiple times, and each time I left, I felt like I should have gone back and begged for a job. Finally, I did just that and had a whirlwind interview with Jayshree after which I was offered a job. There's something special happening there, and the people I listed in the beginning of the chapter are at the heart of it.

Some seven or eight years later, I still see the same level of "wow" almost daily. Arista was a much smaller company then, and when I was hired in 2012, I was just shy of Arista's 600th employee. Arista now has thousands of employees and continues to grow like crazy, and though all companies change as they grow, I'm still floored by some of the things I can do on an Arista switch. In fact, I joke with my students all the time that after they get used to the power of EOS, they'll be frustrated and annoyed using anything else.

Now that I've been an employee for six years, I can still say that the executives at Arista constantly surprise me because they do things like respond well to logic. As someone who's dealt with a lot of executives in my career, it's my impression that most of them find politics and game-playing more important than technical or business acumen. I have never felt that from the Arista executive team. Not only that, but I routinely see C-level executives respond in technical threads internally with not only respectful grace and aplomb, but with a terrific application of deep and accurate technical knowledge. In a world in which I am convinced that most companies survive despite their own best efforts to the contrary, Arista manages to continue to impress me year after year. No company is perfect, and Arista isn't, either, but it is

easily the best place I've ever worked. It is my fervent hope that Arista can continue to impress because having been spoiled by this experience, I don't think I could go back.

Chapter 2. Network Designs

Arista changed the world of networking in more ways than just groundbreaking new products. Seeing a change in the needs of large-scale data centers, Arista promoted the use of some new network design scenarios that have become the standard for many of the Cloud Titans. Those designs have become so successful that they've been adopted by many smaller networks, as well. Why? Mostly because the old designs were laden with complexity as a result of their introduction decades ago when there was no way to avoid such things. Modern networks are much more powerful and, as a result, can be much simpler, though in my experience there is no end to the complexity that a determined network engineer can add.

Remember that these are design principles and they might not apply to every situation. Still, using these designs as a starting point can help immensely when designing new layouts from scratch. Oh, and you'll never see me use terms like *greenfield* unless it's ironically, because I consider that business-speak, and if there's one thing that will make me cross-eyed, it's a nice dose of business-speak. I'd like to spitball a greenfield design so we can shift the paradigm in order to leverage our core competencies in a forward-thinking way that's a win-win solution without boiling the ocean. We'll all need to give 110% as we take it to the next level so we can maximize impact while penetrating multiple demographics. It is what it is.

George Orwell would be proud.

At any rate, there was a valid need for rethinking network designs because the commonly used three-tier architecture was conceived when the majority of the traffic into and out of the data center was what is commonly called *north-south*. Remember the whole core-distribution-access design? That was in three tiers because a request would come in from the top, go down *south* to the next layer where it would be answered, and then sent down a layer to be distributed down to the final layer where it would be worked on, after which the reply would be sent back *north* up the tiers and back to the source. In a world in which routers were expensive standalone devices that were separated by firewalls and switches (and where virtual machines did not yet exist) this three-tier architecture made a lot of sense.

Decades later, almost everything has changed, so the network designs had to change, too. Let's take a look at some of the network designs that are commonly recommended by Arista.

Bow Tie MLAG

When it comes to phrases that make me squirm, one that's high on my list is *Bow Tie MLAG*. MLAG is the open-standard term for Multi-Chassis Link Aggregation Group (see [Chapter 18](#)), but that's not the part of the phrase that bugs me. Yes, I know, the design looks like a bow tie. You know, if you remove some of the links and apply a bit of imagination. Maybe it's just my dislike of bow ties (or any tie, really) that offends me. I therefore propose a new name. How about, *It's not a freaking bow tie MLAG*? Too long? *Black Widow MLAG*? Honestly,

I've got nothing better than Bow Tie MLAG, so I guess it stays. If you see me mumbling incoherently in a corner somewhere, it's probably due to me having to say *Bow Tie MLAG* repeatedly.

Figure 2-1 shows a very simple drawing of a Bow Tie MLAG. This is a Layer 2 design where the top two switches form an MLAG pair and the bottom two also form an MLAG pair. Either pair can be from another vendor's solution such as Cisco's VPC, but so long as there's MLAG involved, we can use the delightful moniker Bow Tie MLAG with reckless abandon. For more information on MLAGs and how to create your own Bow Tie (grumble), see Chapter 18.

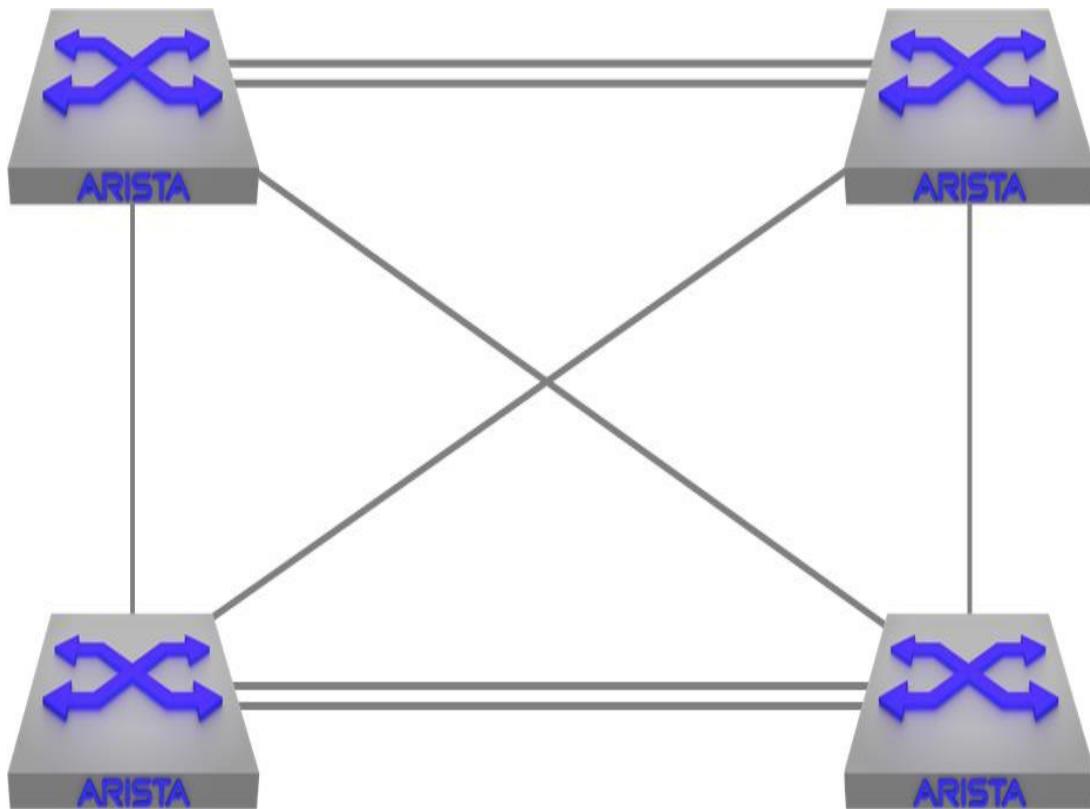


Figure 2-1. Bow Tie MLAG

Leaf-Spine

You might hear this referred to as *Spine-Leaf* or *Leaf-Spine* depending on who's saying or writing it. Regardless of how it's referenced, there are Spines, and there are Leaves, though that's the pedant in me needing to pluralize the word *leaf* as *leaves*. When speaking with network engineers, you might hear the term *leafs*, which I can't decide is an application of plurality for a new word that has nothing to do with trees, or if it's wrong because the plural of the word *leaf* has always been leaves. This is just one more example of why I can't sleep most nights.

This design is based on something called a *Clos* network, which was developed in the 1950s by Charles Clos. The principles of the Clos network have been used in telephony for decades and are also used in Application-Specific Integrated Circuit (ASIC) design. The Clos design principles were used in technologies such as Inifiniband where director switches contained a two-level fat tree (Clos-3) network internally, which included the concept of *Leaf modules* and *Spine modules*. Arista pioneered the terms for use in Ethernet starting around 2008 or so, and the rest, as they say, is history.

Figure 2-2 shows a simplified Leaf-Spine architecture with four Spine switches on the top of the drawing and six Leaf switches on the bottom.

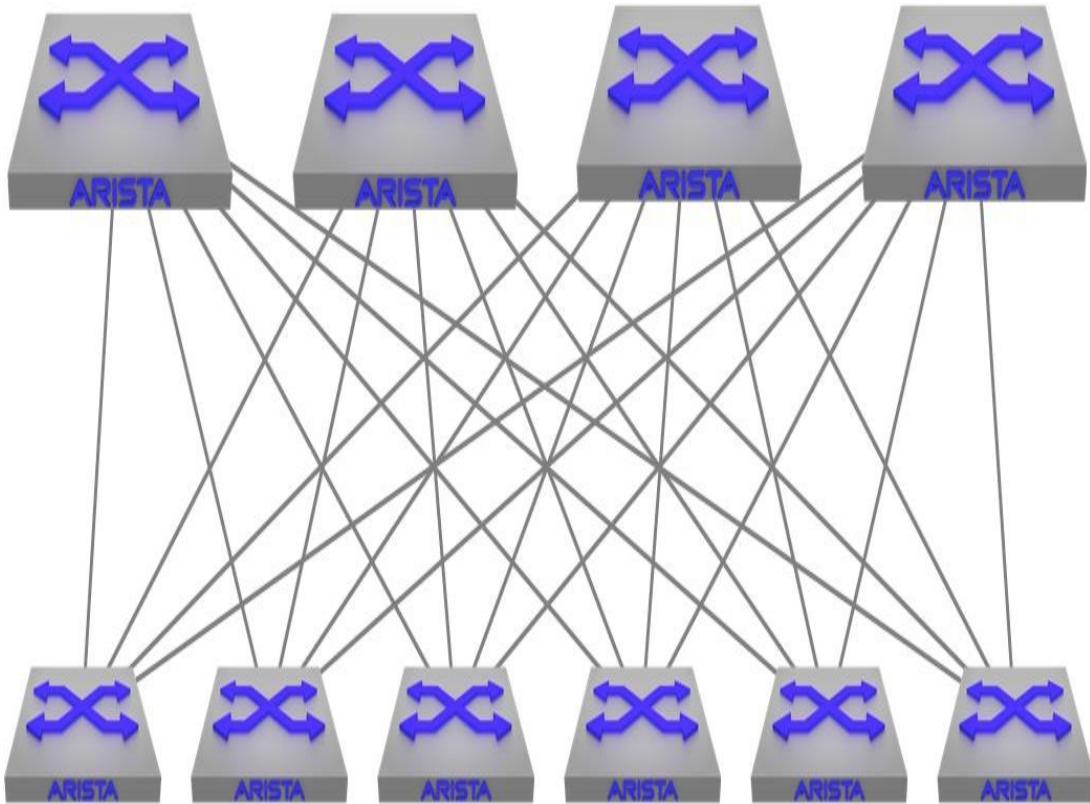


Figure 2-2. Spine-Leaf network design

Originally, Spine switches were larger and more powerful and were analogous to the older core switches in the three-tier architecture in that they're the way that all other devices connect to one another, but there's a lot more to it than that, which is why Arista doesn't call them cores. These days, spine switches are generally just higher-density devices thanks to the advent of devices like 32-port 100 Gbit switches.

Leaf switches are commonly top-of-rack (ToR) switches, and they are connected to all of the Spines. This creates a type of fabric between the Leaf and Spine layers where the distance from any ToR switch to any other ToR switch is only one hop. Not only that, but the Spine layer can suffer many device outages while still maintaining that one-hop distance.

Years ago, there was a lot of talk of protocols like TRansparent Interconnection of Lots of Links (TRILL), which was a way of creating a similar network using Layer 2 (L2) protocols. The Leaf-Spine architecture is different in that it is generally built using Layer 3 (L3) protocols such as Open Shortest Path First (OSPF) or Intermediate System to Intermediate System (IS-IS), but Border Gateway Protocol (BGP) seems to be the winner for a variety of reasons.

Why L3 instead of L2? There used to be a rule in networking that said, “switch when you can; route when you have to.” This rule existed because switching was fast and routing was slow, but with the advent of route forwarding being done in ASIC hardware, that rule is no longer valid. In fact, I’ve been to many customers where they have moved away from L2 entirely opting instead to do L3 right down to the server.

This is all made possible through the addition of something called Equal-Cost MultiPathing (ECMP). With ECMP enabled using BGP (for example), not only are their four paths from each Leaf switch, but all of them are equally desirable through the use of hashing. Now, not only are there many paths for redundancy, but those paths are all active making the entire Leaf-Spine architecture resilient while also being highly utilized. This was (and still is) a game-changer in the world of Ethernet networking because the limitation of a single best path had been significantly improved. As you’ll see, this concept is also expanded upon in other designs.

How many Spines can you have in a Leaf-Spine network? That depends on the hardware in use. Looking at the Arista 7500R switch

data sheet, it is capable of 128-way ECMP, so each Leaf could connect to 128 Spine switches. That's a crazy-wide topology, and if you do the math, that's a lot of math. Luckily Arista sells 16-slot 7500Rs that support 2,304 10 Gbps ports, so that should be enough ports. For now.

NOTE

I joke about 2,304 ports being enough *for now* because Arista continues to blow me away with escalating hardware capacities, so it wouldn't surprise me to see that number increase in ways that I couldn't imagine. I remember buying one of the first Kalpana switches with eight ports of 10 Meg (MEG!) Ethernet. When I wrote the first edition of *Arista Warrior*, 40 Gbps was a big deal and now there are Arista switches that support over 500 ports of 100 Gbps. 400 Gbps and 800 Gbps are now the new hotness. I can only write so fast!

You might see the terms Leaf and Spine used when people talk about MLAG designs, and that's because anything on the top layer has generically become a Spine, whereas anything on the bottom layer has generically become a Leaf. Whether this is correct doesn't really matter so long as whoever is communicating is making sense.

Spline

In the old days, we had something called a *collapsed core* design in which the core and distribution layers were collapsed because who needs three damn layers of networks, anyway? Thinking in a similar vein, the concept of collapsing the Spine and Leaf layers together is called a Spline network, which you can see in Figure 2-3—and, yes, Arista has trademarked that term, no doubt having been collectively annoyed that the entire industry has adopted the phrase Leaf-Spine

without giving them credit.

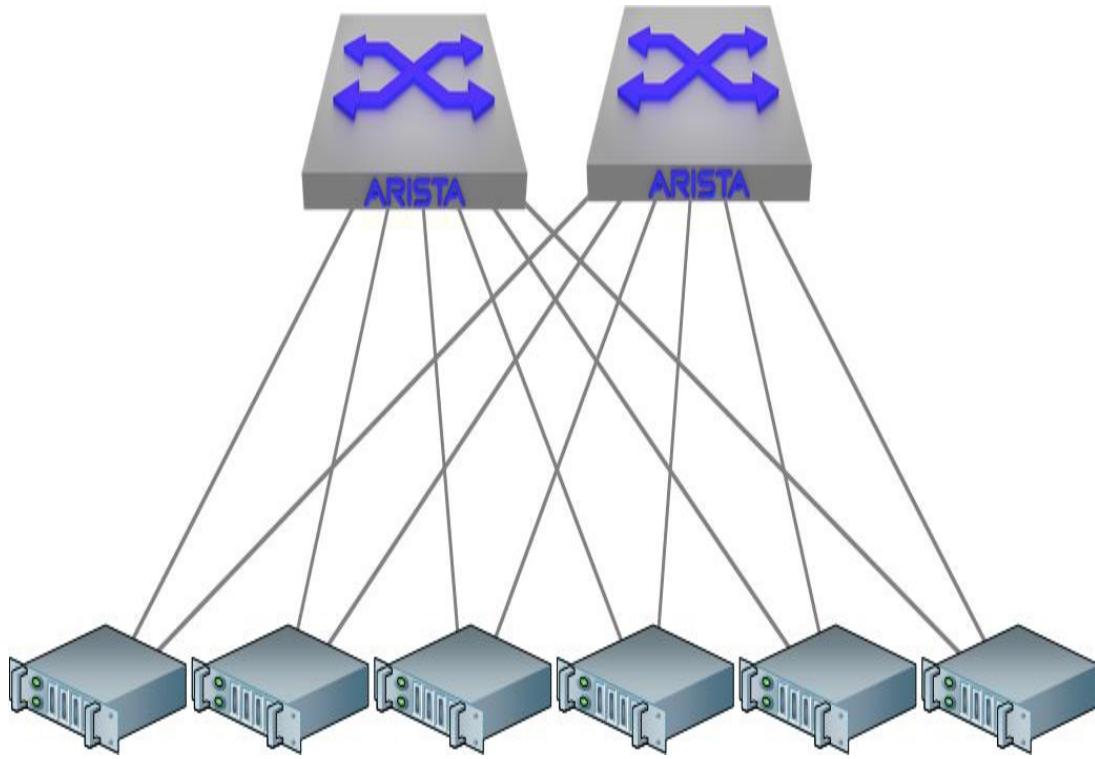


Figure 2-3. Spline network

A Spline network is basically one in which all of the servers (and whatever else) connect to two or more Arista switches. It's a Leaf-Spine network without any Leafs, or to be more accurate, the Spine switches (which don't usually connect to servers) are connecting to servers and are therefore also acting as Leafs. As one of my favorite tech-reviewers commented, "That sounds stupid," which made me laugh so hard that I needed to keep it while also commenting on it. Consider this design to be more like an End-of-Row (EoR) design in which all of the servers in the row connect to a pair of switches at the end of the row. Anyway, because we've mashed the Leafs and Spines together, we get the portmanteau Spline.

A Spline network is probably more of an enterprise type of design,

though there's no reason it can't be used anywhere that the scale of a full Leaf-Spine topology is unnecessary or not cost effective.

VXLAN/Overlay

Virtual Extensible LAN (VXLAN) (covered in [Chapter 21](#)) is a means of building an L2 *overlay* network on top of an existing L3 *underlay* network. Essentially L2 tunnels are built on top of an existing L3 topology. In the case of [Figure 2-4](#), there are two VXLAN Segments shown (dotted lines) riding on top of the existing Leaf-Spine network.

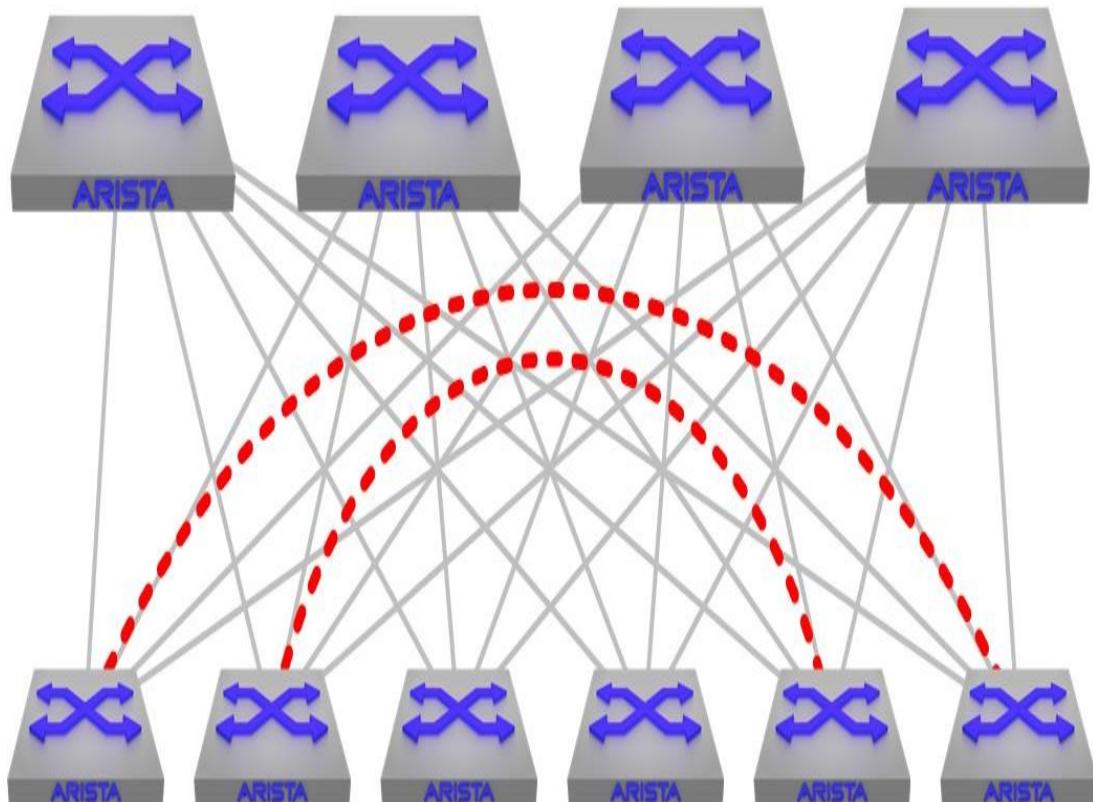


Figure 2-4. VXLAN overlay network

For reasons outlined in [Chapter 21](#), this design is becoming more and more common. I've seen it within rows, within data centers, between data centers, and even over the internet (though that didn't work well

for them), so it's pretty much everywhere these days.

Universal Spine

All of these network designs are great, but what if you have a complex network spanning multiple locations, or even multiple rows in a massive data center, each with its own potentially different network design? Enter the Universal Spine.

Figure 2-5 shows each of the previously mentioned network designs along the bottom with each of them connected to a new Universal Spine network, in this case comprising Arista Chassis switches. And yes, it took me a long damn time to connect all those links.

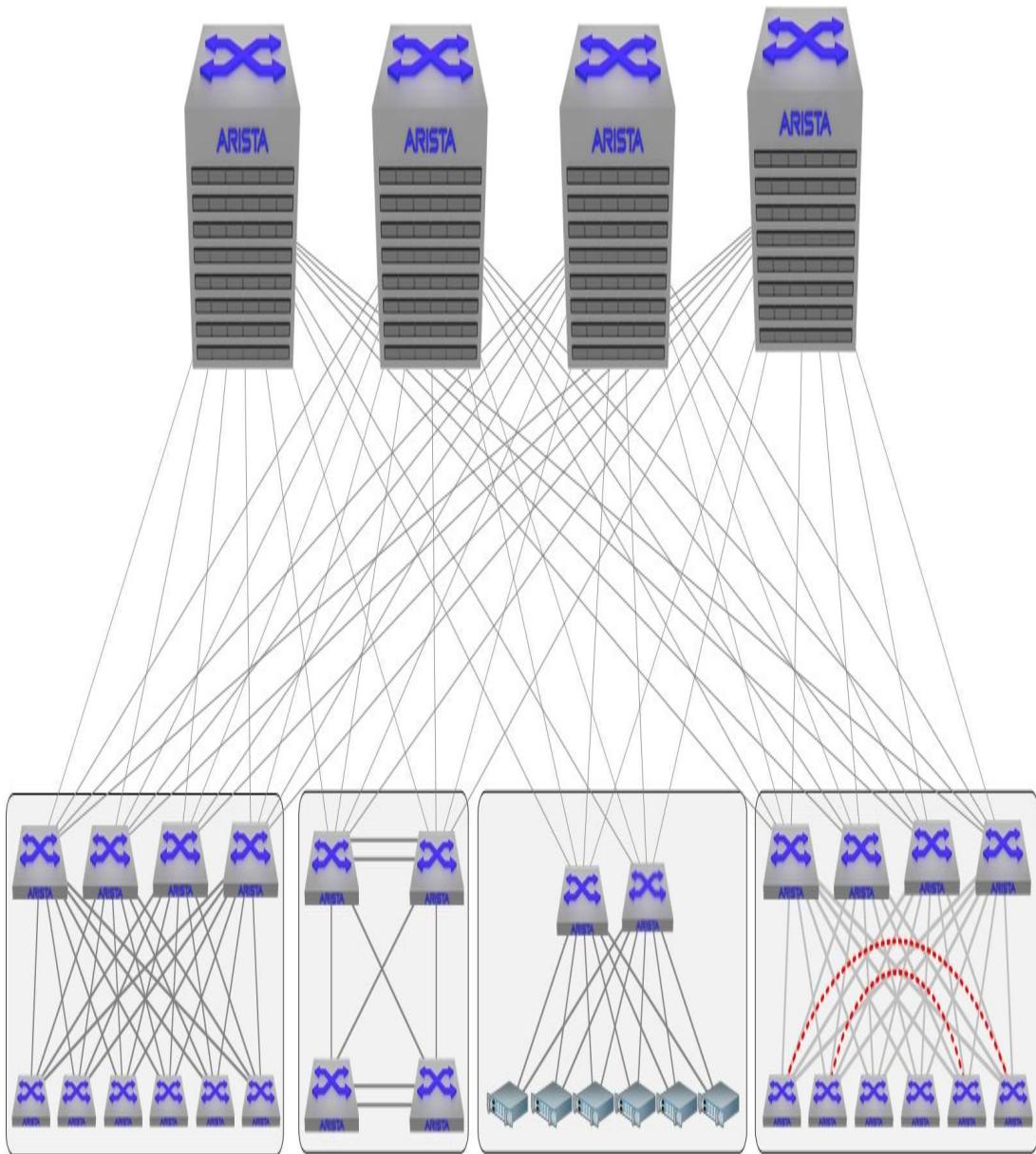


Figure 2-5. Universal Spine

With the ability to do wire-speed switching and routing in hardware and the ability to support massive route tables and huge ECMP, the Universal Spine allows the connection of disparate network topologies with one another and to the rest of the world, as well. Don't forget that this Universal Spine can also be connecting any number of cloud solutions including public and private clouds with services like Amazon Web Services (AWS), Microsoft Azure, and the like, all

competing for your networking dollars.

AnyCloud

AnyCloud is essentially the ability to connect your Arista networks to a cloud-based environment using vEOS-Router while also incorporating security through a Virtual Private Network (VPN). With the widespread adoption of solutions like AWS and Microsoft Azure, many networks are now fully involved in something Arista calls the *Hybrid Cloud architecture*. Your network might have local devices, cloud devices, virtual cloud devices, and who knows what else. AnyCloud allows you to connect them all. Of course, you can manage it all with Cloudvision, too.

Conclusion

These descriptions are clearly only high-level overviews of each of the topologies listed. Details of how to build them all is spread throughout this book, but having an understanding of what each of these designs was created for will help you when you're discussing design scenarios for the modern data center, campus, or enterprise network.

Chapter 3. Buffers

When you begin talking to vendors about data center switches, you'll hear and read about buffers. Some of the vendors have knockdown, drag-out fights about these buffers, and often engage in all sorts of half-truths and deceptions to make you believe that their solution is the best. So, what is the truth? As with most things, it's not always black and white.

To begin, we need to look at the way a switch is built. That starts with the switch fabric.

NOTE

The term *fabric* is used because the interconnecting lines of ports and switches evokes the weave of a fabric viewed through a microscope. And all this time I thought there was some cool scientific reason.

Imagine a matrix in which every port on the switch has a connection for input (ingress) and another for output (egress). If we put all of the ingress ports on the left and all the output ports on top and then interconnect them all, it would look like the drawing in Figure 3-1. To make the examples easy to understand, I've constructed a simple, though thoroughly unlikely, three-port switch. The ports are numbered ethernet1, ethernet2, and ethernet3, which are abbreviated e1, e2, and e3.

Looking at the drawing, remember that e1 on the left and e1 on the top are *the same port*. This is very important to understand before moving forward. Remember that modern switch ports are generally full duplex. The drawing simply shows the *ins* on the left and the *outs* on the top. Got it? Good. Let's continue.

First, the fabric allows more than one conversation to occur at a time, provided the ports in each conversation are discrete from the ports in the other conversations. I know, gibberish, right? Bear with me, and all will become clear.

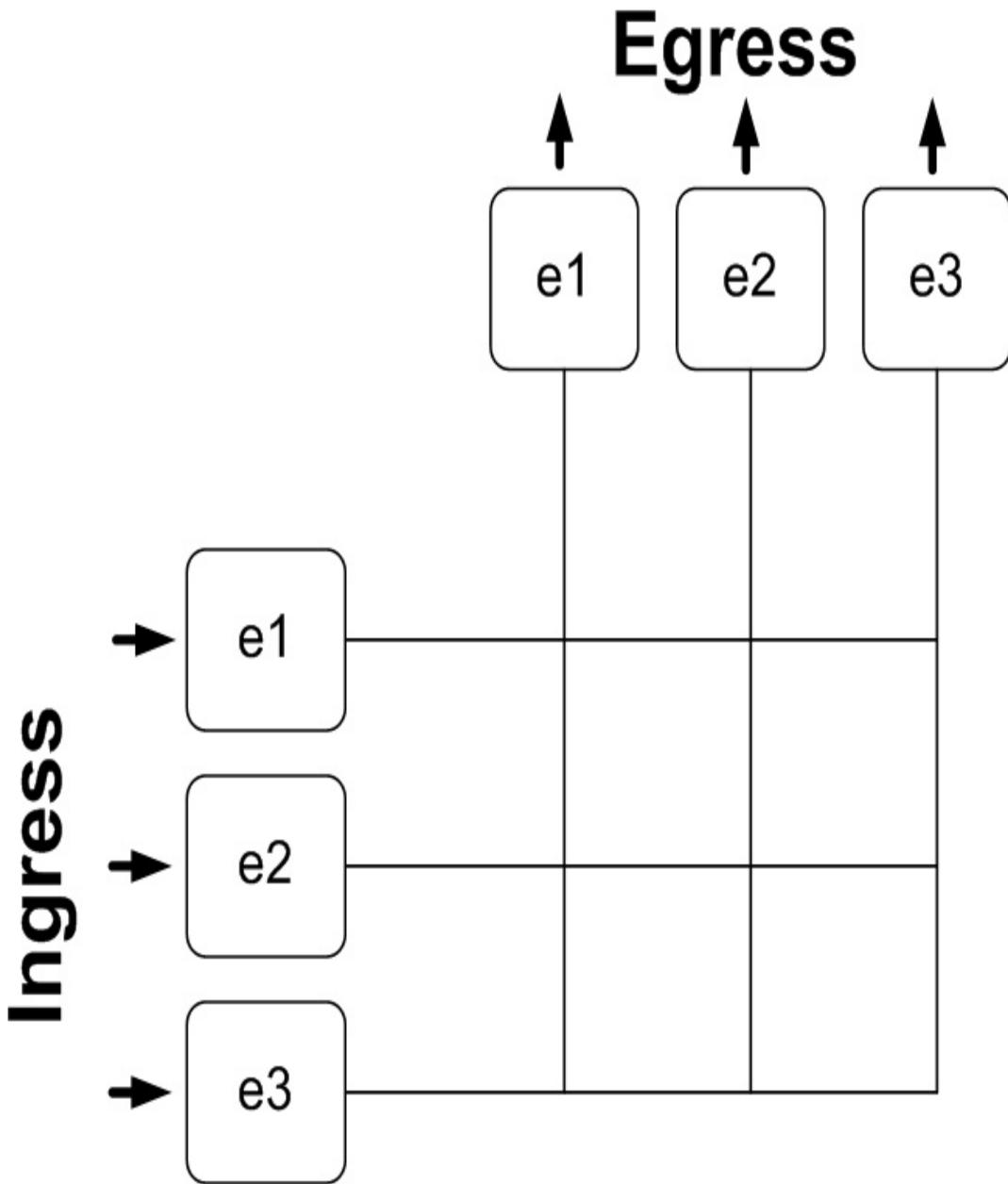


Figure 3-1. Simple switch fabric of a three-port switch

Remember that full duplex means transmit and receive can happen at the same time between two hosts (or ports, in our case). To help solidify how the fabric drawing works, take a look at Figure 3-2, in which I've drawn up how a full-duplex conversation would look between ports e1 and e2.

Look at how the input of e1 goes to the point on the fabric where it can traverse to the output of e2. Now look at how the same thing is happening so that the input of e2 can switch to the output of e1. This is what a full-duplex conversation between two ports on a switch looks like on the fabric. By the way, you should be honored, because I detest those little line jumpers and haven't used one in probably 10 years. I have a feeling that this chapter is going to irritate my drawing sensibilities, but I'll endure because I have deadlines to meet, and after staring at the drawings for two hours, I couldn't come up with a better way to illustrate my point.

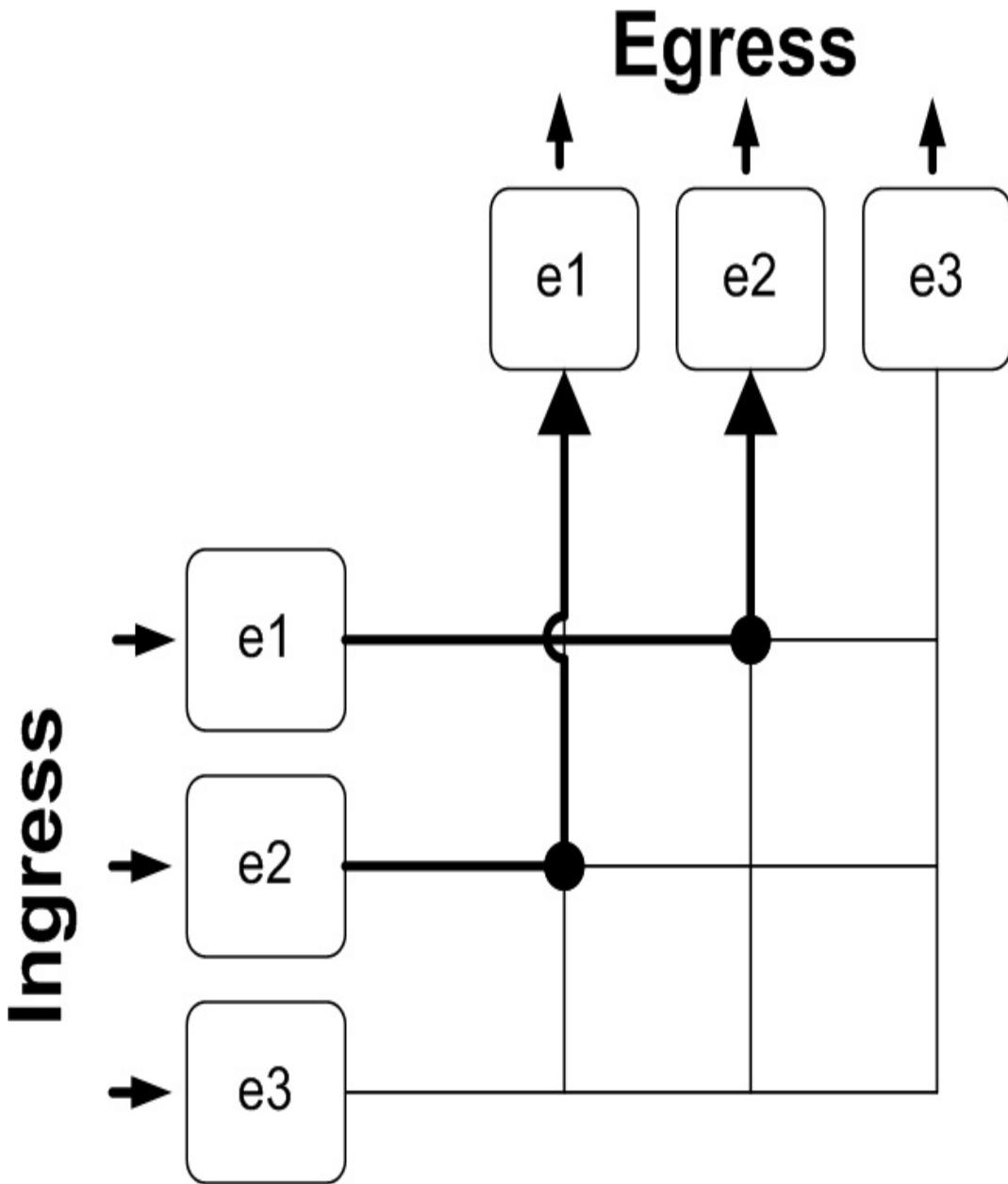


Figure 3-2. Full duplex on a switch fabric

Now that we know what a single port-to-port full-duplex conversation looks like, let's consider a more complex scenario. Imagine if you will, that while ports e1 and e2 are happily chattering back and forth without a care in the world, some jackass on e3 wants to talk to e2. Because Ethernet running in full duplex does not listen for traffic before transmitting, e3 just blurts out what he needs to say. Imagine that you

are having a conversation with your girlfriend on the phone when your kid brother picks up the phone and plays death metal at full volume into the phone. It's like that, but without the heavy distortion, long hair, and tattoos.

Assuming for a moment that the conversation is always on between e1 and e2, when e3 sends its message to e1, what happens? In our simple switch, e3 will detect a collision and drop the packet. Wait a minute, a collision? I thought full-duplex networks didn't have collisions! Full-duplex conversations should not have collisions, but in this case, e3 tried to talk to e2 and e2 was busy. That's a collision. Figure 3-3 shows our collision in action. The kid brother is transmitting on e3, but e2's output port is occupied, so the death metal is dropped. If only it were that simple in real life.

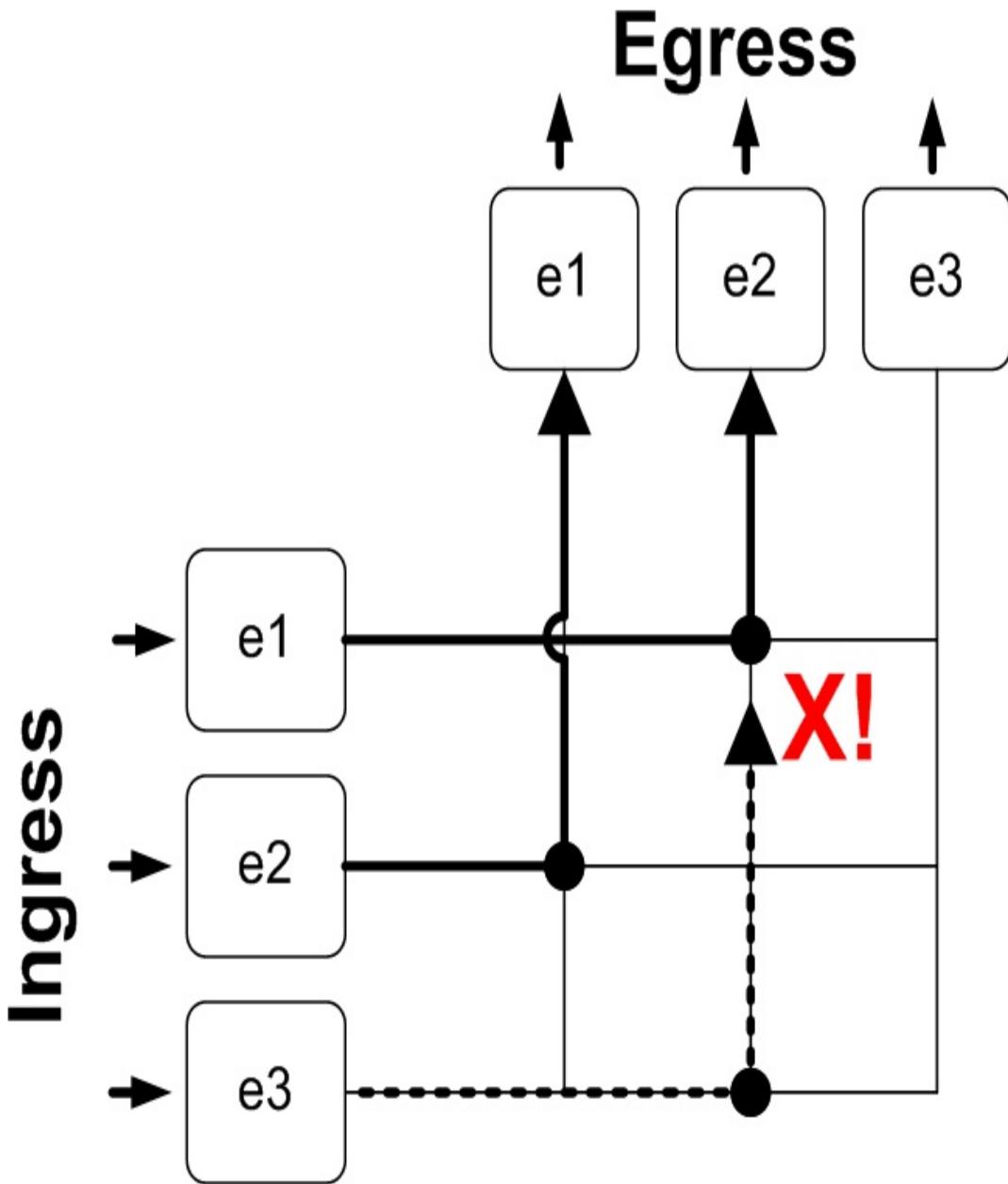


Figure 3-3. Switch fabric collision

If you think that this sounds ridiculous and doesn't happen in the real world, you're almost right. The reason it doesn't seem to happen in the real world, though, is largely because Ethernet conversations are rarely always on, and because of buffers.

In Figure 3-4, I've added input buffers to our simple switch. Now,

when port e3 tries to transmit, the switch can detect the collision and buffers the packets until the output port on e2 becomes available. The buffers are like little answering machines for Ethernet packets. Now, when you hang up with your girlfriend, the death metal can be politely delivered in all its loud glory now that the output port (you) is available. God bless technology.

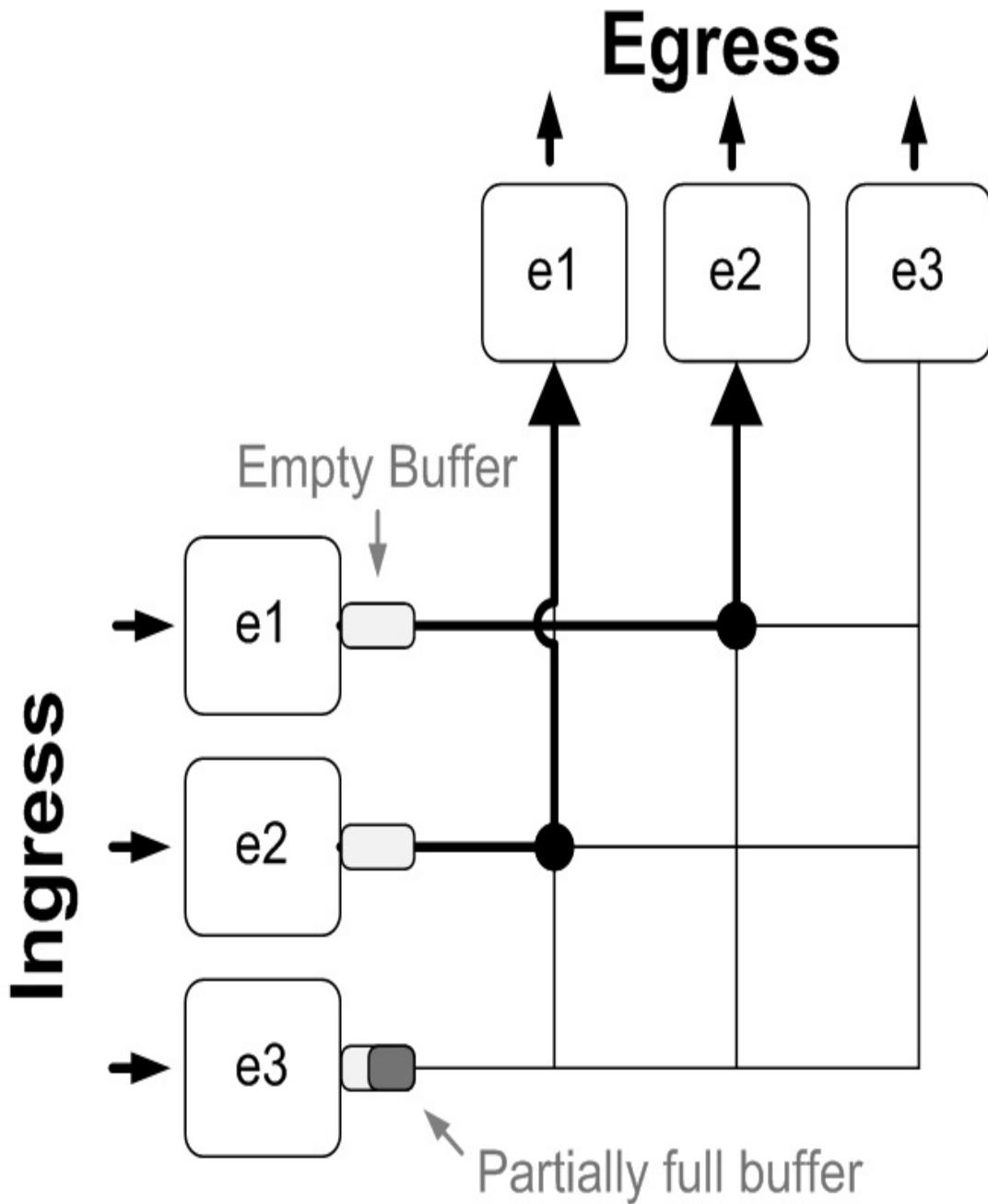


Figure 3-4. Switch fabric with input buffers

This is cool and all, but these input buffers are not without their limitations. Just as an answering machine tape (anyone remember those?) or your voicemail inbox can fill up, so too can these buffers. When the buffers become full, packets are dropped. Whether the first packets in the buffer are dropped in favor of buffering the newest

packets or the newest packets are dropped in favor of the older packets is up to the person who wrote the code.

So, if the buffers can fill up, thus dropping packets, the solution is to put in bigger buffers, right? Well, yes and no. The first issue is that buffers add latency. Sending packets over the wire is fast. Storing packets into a location in memory and then referencing them and sending them takes time. Memory is also slow, although the memory used in these buffers is much faster than, say, computer RAM. It's more like the Layer 2 (L2) cache in your CPU, which is fast, but the fact remains that buffering increases latency. Increased latency is usually better than dropped packets, right? As usual, it depends.

Dropped packets might be OK for something like FTP that will retransmit lost packets, but for a UDP-RTP stream like VoIP, increased latency and dropped packets can be disastrous. And what about environments like Wall Street, where microseconds of latency can mean a missed sale opportunity costing millions of dollars? Dropped packets mean retransmissions, which means waiting, but bigger buffers still mean waiting—they just mean waiting less. In these cases, bigger buffers aren't always the answer.

In the example I've shown, I started with the assumption that the full-duplex traffic to and from e1 and e2 is always on, which is almost never the case. In reality, Ethernet traffic tends to be very bursty, especially when there are many hosts talking to one device. Consider scenarios like email servers, or even better, NAS towers.

Network Attached Storage (NAS) traffic can be unpredictable when

looking at network traffic. If you have 100 servers talking to a single NAS tower on a single IP address, the traffic to and from the NAS tower can spike in sudden, drastic ways. This can be a problem in many ways, but one of the most insidious is the *microburst*.

A microburst is a burst that doesn't show up on reporting graphs because most sampling is done using five-minute averages. If a monitoring system polls the switch every five minutes and then subtracts the number of bytes (or bits, or packets) from the number reported during the last poll, the resulting graph will show only an average of each five-minute interval. Because pictures are worth 1,380 words (adjusted for inflation), let's take a look at what I mean.

In Figure 3-5, I've taken an imaginary set of readings from a network interface. Once, every minute, the switch interface was polled and the number of bits per second was determined. That number was recorded with a timestamp. If you look at the data, you'll see that once every 6 to 10 minutes or so, the traffic spikes 50 times its normal value. These numbers are pretty small, but the point I'm trying to make is how the reporting tools might reveal this information.

The graph on the top shows each poll, from each minute, and includes a trend line. Note that the trend line is at about 20,000 bits per second (bps) on this graph.

Time	1 min bps	Time	5 min bps
10:00	2,000	10:05	112,000
10:01	2,000	10:11	134,000
10:02	103,000	10:17	102,000
10:03	2,000	10:23	119,000
10:04	2,000	10:29	112,001
10:05	1,000		
10:06	3,000		
10:07	2,000		
10:08	120,000		
10:09	2,000		
10:10	3,000		
10:11	4,000		
10:12	2,000		
10:13	2,000		
10:14	3,000		
10:15	90,000		
10:16	2,000		
10:17	3,000		
10:18	1,000		
10:19	2,000		
10:20	2,000		
10:21	110,000		
10:22	1,000		
10:23	3,000		
10:24	2,000		
10:25	1,000		
10:26	2,000		
10:27	4,000		
10:28	100,001		
10:29	3,000		
10:30	2,000		

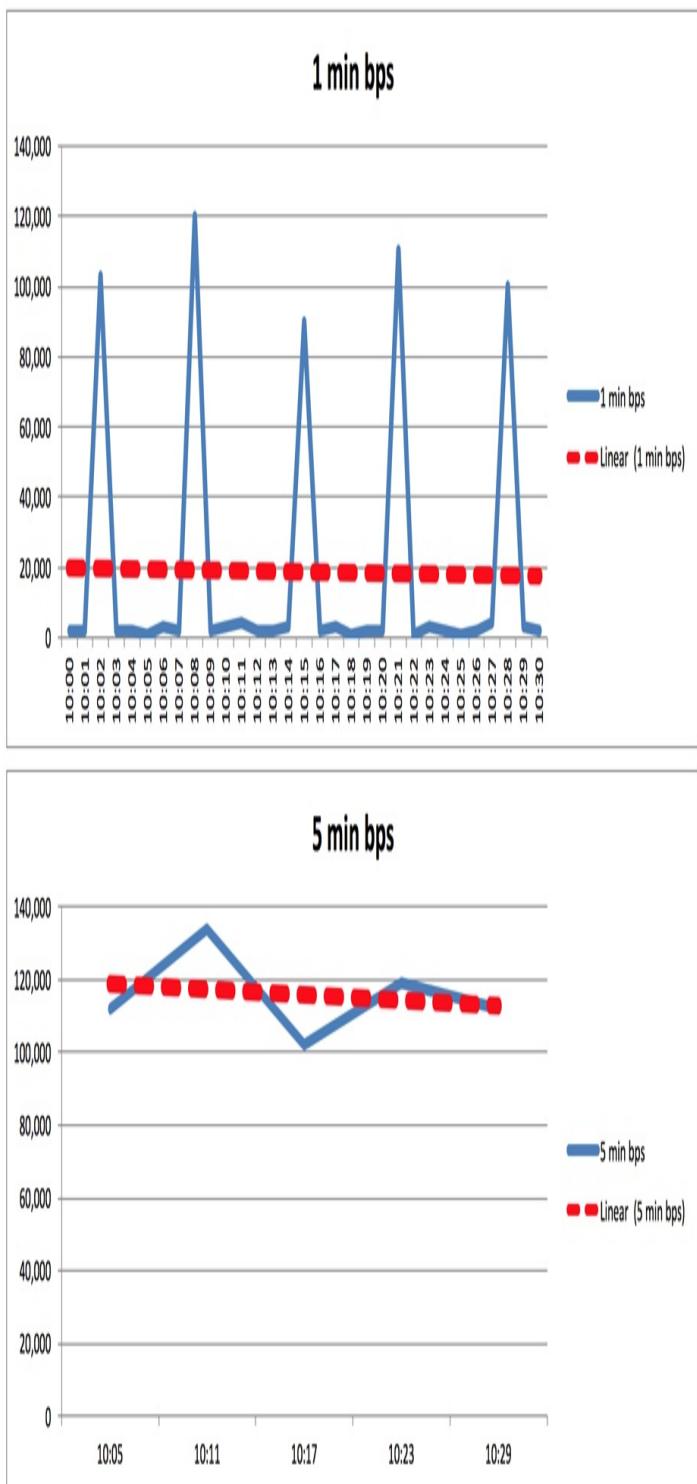


Figure 3-5. Microbursts and averages

Now take a careful look at the bottom graph. In this graph, the data looks very different because instead of including every one-minute poll, I've changed the polling to once every five minutes. In this graph, the data seems much more stable and doesn't appear to show any sharp spikes. More important, though, is the fact that the trend line seems to be up at around 120,000 bps.

This is typical of data being skewed because of the sample rate, and it can be a real problem when the perception doesn't meet reality. The reality is closer to the top graph, but the perception is usually closer to the bottom graph. Even the top graph might be wrong, though! Switches operate at the microsecond or even nanosecond level. So, what happens when a 10 Gbps interface has 15 Gbps of traffic destined to it, all within a single second or less? Wait, how can a 10 Gbps interface have more than 10 Gbps being sent to it?

Remember the fabric drawing in [Figure 3-3](#)? Let's look at that on a larger scale. As referenced earlier, imagine a network with 100 servers talking to a single NAS tower on a single IP address. What happens if, say, 10 of those servers push 5 Gbps of traffic to the NAS tower at the same instance in time? The switch port connecting to the NAS switch will send out 10 Gbps (because that is the max), and 40 Gbps of traffic will be queued.

Network switches are designed to forward packets (frames, to be pedantic) at the highest rate possible. Few devices outside of the networking world can actually send and receive data at the rates the networking devices are capable of sending. In the case of NAS towers, the disks add latency, the processing adds latency, and the OS of the

device simply might not be able to deliver a sustained 10 Gbps data stream. So, what happens when our switch has a metric pant-load of traffic to deliver and the NAS tower can't accept it fast enough?

If the switch delivers the packets to the output port but the attached device can't receive them, the packets will again be buffered, but this time as an output queue. [Figure 3-6](#) shows our three-port switch with output buffers added.

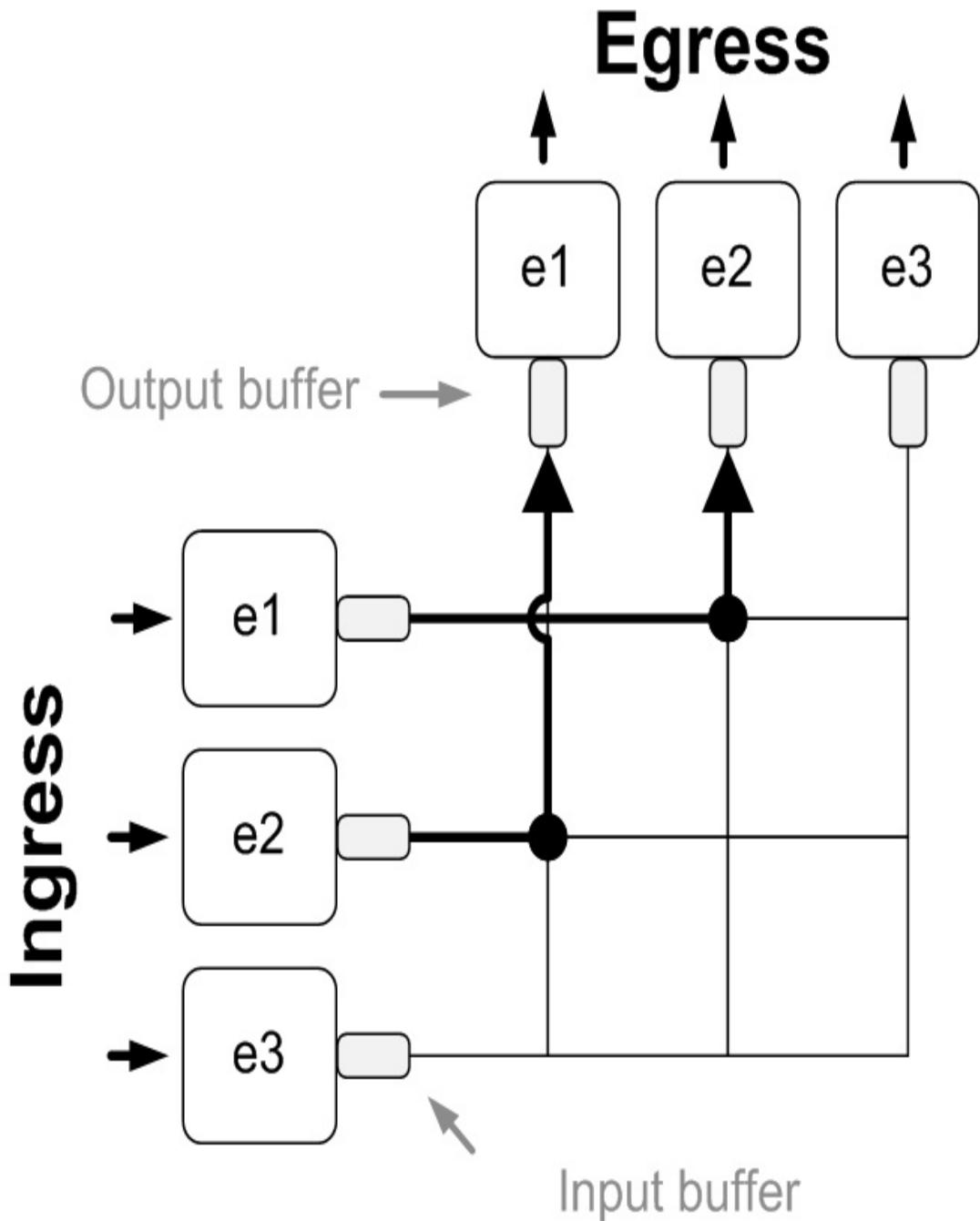


Figure 3-6. Switch fabric with output buffers

As you might imagine, the task of figuring out when traffic can and cannot be sent to and from interfaces can be a complicated affair. It was simple when the interface was either available or not, but with the addition of buffers on both sides, things become more complicated, and

this is an extreme simplification. Consider the idea that different flows might have different priorities, and the entire affair becomes even more complicated.

The process of determining when, and if, traffic can be sent to an interface is called *arbitration*. Arbitration is usually managed by an Application-Specific Integrated Circuit (ASIC) within the switch and generally cannot be configured by the end user. Still, when shopping for switches, some of the techniques used in arbitration will come up, and understanding them will help you decide what to buy. Now that we understand why input and output buffers exist, let's take a look at some terms and some of the ways in which traffic is arbitrated within the switch fabric:

FIFO

First In/First Out buffers are those that deliver the oldest packets from the buffer first. When you drive into a tunnel and the traffic in the tunnel is slow, assuming no change in the traffic patterns within the tunnel, the cars will leave the tunnel in the same order in which they entered: the first car into the tunnel will also be the first car out of the tunnel.

Blocking

Blocking is the term used when traffic cannot be sent, usually due to oversubscription. A nonblocking switch is one in which there is no oversubscription, and each port is capable of receiving and delivering wire-rate traffic to and from another interface in the switch. If there are 48 10 Gb interfaces and the switch has a fabric speed of 480 Gbps (full duplex), the switch can be said to be nonblocking, but be careful because some vendors will be less than honest about these numbers. For example, stating that a 48-port 10

Gb switch has a 480 Gbps backplane does not necessarily indicate that the switch is nonblocking, because traffic can flow in two directions in a full-duplex environment. 480 Gbps might mean that only 24 ports can send at 10 Gbps, whereas the other 24 receive at 10 Gbps. This would be 2:1 oversubscription to most people, but when the spec sheet says simple 480 Gbps, people assume the best. Clever marketing and the omission of details like this are more common than you might think.

Head-of-Line (HOL) blocking

Packets can be (and usually are) destined for a variety of interfaces, not just one. Consider the possibility that with the FIFO output queue on one interface, packets will buffer on the FIFO input buffer side. If the output queue cannot clear quickly enough, the input buffer will begin to fill, and none of those packets will be switched, even though they might be destined for other interfaces. This single packet, sitting at the head of the line, is preventing all of the packets behind it from being switched, as shown in [Figure 3-7](#). Using the car analogy, imagine that there is a possible left turn directly outside the end of the tunnel. It's rarely used, but when someone sits there, patiently waiting for a break in oncoming traffic, everyone in the tunnel must wait for this car to move before they can exit the tunnel.

NOTE

If you're reading this in a country that drives on the left side of the road, please apply the following regular expression to my car analogies as you read: `s/left/right/g`.
Thanks.

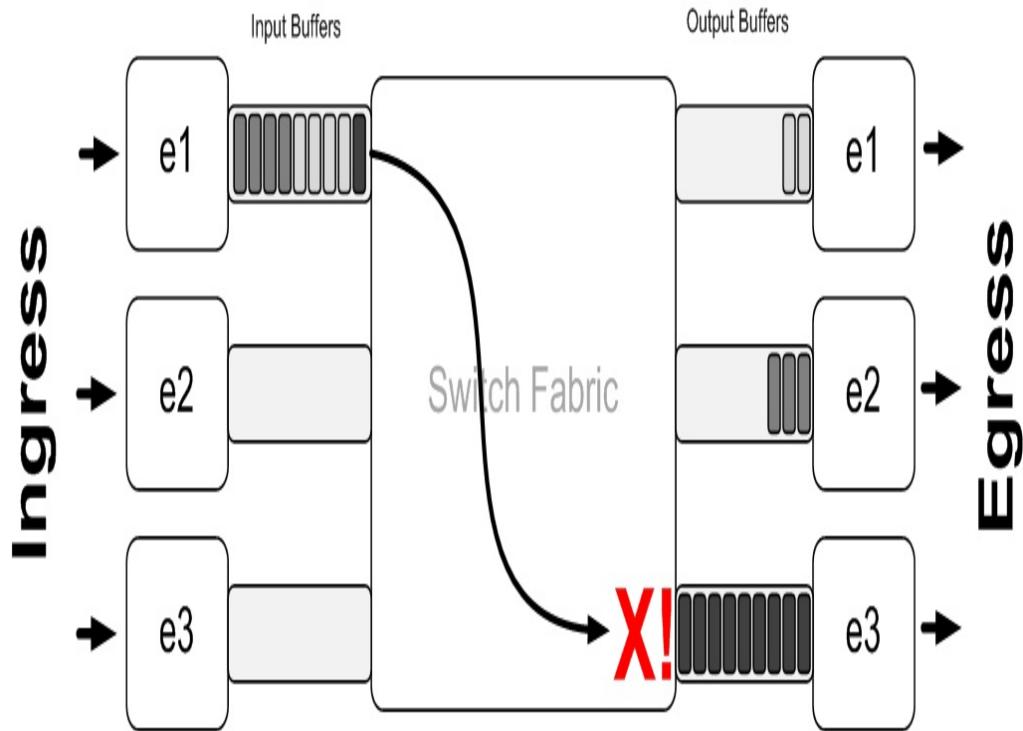


Figure 3-7. Head-of-line blocking

Virtual Output Queuing (VOQ)

VOQ is one of the methods deployed by switch vendors to help eliminate the HOL blocking problem on their higher-end switch (shown in Figure 3-8). If there were a buffer for each output interface, positioned at the input buffer side of the fabric and replicated on every interface, HOL blocking would be practically eliminated.

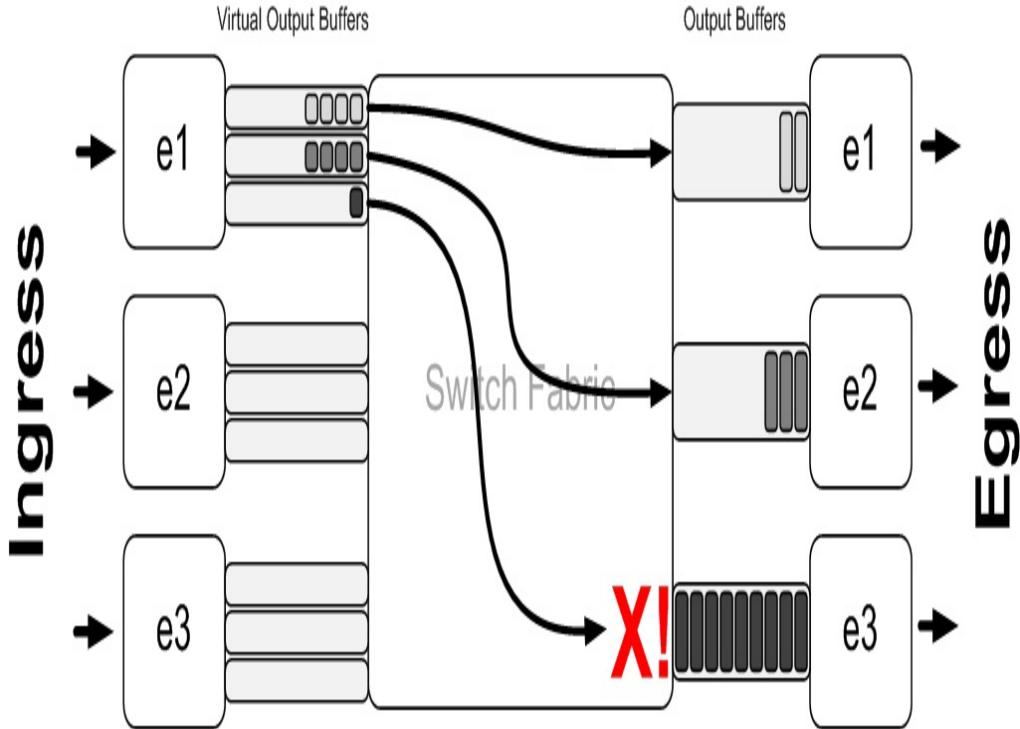


Figure 3-8. Virtual Output Queuing

Now, because there is a virtual output queue for every interface on the input side of the fabric, should the output queue become full, the packets destined for the full output queue will sit in its own virtual output queue, leaving the virtual output queues for all of the other interfaces unaffected. In our left-turn-at-the-end-of-the-tunnel example, imagine an additional left-turn-only lane being installed. While the one car waits to turn left, the cars behind it can simply pass because the waiting car is no longer blocking traffic.

Allocating a single virtual output queue for each possible output queue would quickly become unscalable, especially on switches with thousands of interfaces. Instead, each input queue can have a smaller set of VOQs, which can be dynamically allocated as needed. The idea is that eight flows is probably more than enough for all but the most demanding of environments.

If you start reading up on buffers elsewhere, you are likely to encounter dire warnings about excessively large buffers, and something colorfully

referred to as *buffer bloat*. Buffer bloat describes the idea that hardware vendors have increasingly included more and more buffers in an attempt to outperform competitors. Even though buffer bloat may be a real concern in the home internet environment, it is likely not a concern in the data center.

Consider what happens when you stream a movie from your favorite streaming source (let's call them Stream-Co). The servers might have 10 Gbps interfaces, which are connected with 10 Gbps switches, and because they're a big provider, they might even have 10 Gbps internet feeds. The internet is interconnected with pretty fast gear these days, so let's say, just for fun, that all the connections from Stream-Co to your ISP network are 10 Gbps. Yeah, baby—fast is good! Now, your cable internet provider switches your stream in 10 glorious gigabits per second, until it gets to the device that connects to your cable modem. Suppose that you have a nice connection, and you can download 50 Mbps. Can you see the problem?

The kickin' 10 Gbps data flow from Stream-Co has screamed across the country (or even the world) until it gets right to your virtual doorstep, at which point the speed goes from 10 Gbps to 50 Mbps. The difference in speed is not 10:1 like it is in a data center switch, but rather 200:1!

Now let's play a bit and assume that the cable distribution device has 24 MB buffers. Remember, that 24 MB at 1 Gbps is 20 ms. Well, that same 24 MB at 50 Mbps is 4 seconds! Buffering for 20 ms is not a big deal, but buffering for 4 seconds will confuse the TCP windowing system, and your performance might be less than optimal, to say the

least. Additionally, although 24 MB is 4 seconds at 50 Mbps, remember that it's only 0.019 seconds at 10 Gbps. In other words, this buffer would take less than one-tenth of a second to fill, but 4 seconds to empty.

Think about this, too: *propagation delay* (the time it takes for packets to travel over distance) from New York to California might be 100 ms over multiple providers. Let's add that much on top for *computational delay* (the amount of time it takes for servers, switches, and routers to process packets), which gives us 200 ms. That's one-fifth of a second, which is a pretty long time in our infinitely connected high-speed world. Imagine that your service provider is getting packets in 200 ms but is buffering multiple seconds of your traffic. To quote some guy I met on the beach in California, that's not cool, man.

My point with this talk of buffer bloat is to consider all the information before coming to rash conclusions. You might hear vendors pontificate about how big buffers are bad. Big buffers within the data center make a lot more sense than big buffers for cable modems.

Conclusion

Deep buffers can be a significant advantage in a networking switch, and because Arista was one of the first vendors to champion that idea, other vendors attacked the idea before themselves embracing it. Between buffer size, not oversubscribing the fabric ([Chapter 5](#)), and the many other features that Arista has embraced ([Chapter 6](#)), it has become a force to be reckoned with in the world of networking.

Chapter 11. Link Layer Discovery Protocol

Link Layer Discovery Protocol (LLDP) is the open standard (IEEE 802.1AB) means of discovering neighbors over the link layer (Layer 2 [L2]). Let's take a look at LLDP in action and see what you might need to do to make it work with other vendors' devices.

To show a bit of intervendor compatibility, I've connected my Arista 7010T switch to an old Cisco 3750. The connection is between G1/0/52 on the 3750 and e24 on the Arista. When I turn up the port, I get the following message on the Arista console:

```
Apr 10 23:21:35 Arista Lldp: %LLDP-5-NEIGHBOR_NEW: LLDP neighbor with
chassisId 001c.b084.cfb4 and portId "[ Arista e24 ]" added on
interface Ethernet24
```

Without configuring anything, the Arista switch has discovered the Cisco switch, even though the Arista switch is not running Cisco Discovery Protocol (CDP). Let's dig in and see what Arista sees:

```
Arista-7010T#sho lldp
LLDP transmit interval      : 30 seconds
LLDP transmit holdtime      : 120 seconds
LLDP reinitialization delay : 2 seconds
LLDP Management Address VRF : default

Enabled optional TLVs:
  Port Description
  System Name
  System Description
```

```

System Capabilities
Management Address (best)
IEEE802.1 Port VLAN ID
IEEE802.3 Link Aggregation
IEEE802.3 Maximum Frame Size

Port      Tx Enabled Rx Enabled
Et1       Yes        Yes
Et2       Yes        Yes
Et3       Yes        Yes
Et4       Yes        Yes
Et5       Yes        Yes
Et6       Yes        Yes
Et7       Yes        Yes
Et8       Yes        Yes
Et9       Yes        Yes
Et10      Yes        Yes

```

[--snippage--]

```

Et45      Yes        Yes
Et46      Yes        Yes
Et47      Yes        Yes
Et48      Yes        Yes
Et49      Yes        Yes
Et50      Yes        Yes
Et51      Yes        Yes
Et52      Yes        Yes
Ma1       Yes        Yes

```

Arista-7010T#

That's pretty boring. I'd rather see what switches are connected where, so let's use the `show lldp neighbors` command:

```

Arista-7010T#sho lld nei
Last table change time  : 73 days, 2:44:38 ago
Number of table inserts : 5
Number of table deletes : 1
Number of table drops   : 0
Number of table age-outs : 0

Port      Neighbor Device ID          Neighbor Port ID          TTL
Et13      VLab-Linux                 0025.9088.2e90          120
Et33      3750-Core.gad.net         [ Arista Core ]          120

```

Now that's more like it! Port e13 is a server called VLab-Linux. On port e33, the Neighbor Device ID is shown as 3750-Core.gad.net, which is a conglomeration of the hostname and the configured domain name on the 3750. If you've never changed the domains on your Cisco switches, expect them all to show up as *hostname.cisco.com*. I detest defaults when my OCD kicks in, so I changed my 3750 to gad.net:

```
SW-3750(config)#ip domain name gad.net
```

Unfortunately, the Cisco switch is not so open-minded, at least by default. Here's the output of the command `show cdp neighbors` on the 3750:

```
SW-3750#sho cdp neighbors
Capability Codes: R - Router, T - Trans Bridge, B- Source Route Bridge
                  S - Switch, H - Host, I - IGMP, r - Repeater, P - Phone

Device ID      Local Intrfce Holdtme   Capability  Platform  Port ID
R1-PBX        Gig 1/0/10    144        R S I      2811      Fas 0/0
R1-PBX        Gig 1/0/11    135        R S I      2811      Fas 0/1
TS-1          Gig 1/0/39    122        R          2611      Eth 0/1
SEP0019AA96D096 Gig 1/0/42    126        H P       IP Phone  Port 1
Cisco-WAP-N   Gig 1/0/1     120        T I       AIR-AP125 Gig 0
SEP04FE7F689D33 Gig 1/0/2     125        H P       IP Phone  Port 1
SEP000DBC50FCD1 Gig 1/0/4     147        H P       IP Phone  Port 1
SEP00124362C4D2 Gig 1/0/42    147        H P       IP Phone  Port 1
```

Although there are all sorts of interesting devices like WAPs and IP Phones listed, there is no mention of the Arista switch. And although the Arista switch will listen to and understand the CDP advertisements, the 3750 doesn't see the LLDP advertisements being sent by Arista. Luckily, we can change that by using the Cisco command `lldp run`:

```
SW-3750(config)#lldp run
```

Even though this won't let us see the Arista switch with the `show cdp neighbor` command, we can now see it with the `show lldp neighbor` command:

```
SW-3750#sho lldp neighbors

Capability codes:
  (R) Router, (B) Bridge, (T) Telephone, (C) DOCSIS Cable Device
  (W) WLAN Access Point, (P) Repeater, (S) Station, (O) Other

Device ID      Local Intf Hold-time  Capability  Port ID
SEP04FE7F689D33  Gi1/0/2    180        B,T        04FE7F689D33:P1
Office Switch  Gi1/0/42    120        B          g1
Arista         Gi1/0/52    120        B          Ethernet24

Total entries displayed: 3
```

Not only can we see the Arista switch on port G1/0/52, but we can also see some other devices that we heretofore could not see. The device named **Office Switch** is a Netgear eight-port 1 Gb switch that I didn't even know supported LLDP. What a pleasant surprise! I also found it interesting that the IP Phone with the device ID of `SEP04FE7F689D33` supports CDP and LLDP simultaneously.

As with CDP on a Cisco switch, you can show detail information. Using the `show lldp neighbors detail` command will output a pile of useful information, but it's displayed a bit differently than the similar Cisco command. Whereas Cisco sorts this information by device discovered, Arista sorts it by interface and shows every interface on the switch. In the interest of brevity, I've removed some of the output, including one of the Arista connected interfaces:

```
Arista(config)# sho lldp neighbors detail
Interface Ethernet1 detected 0 LLDP neighbors:
```

```
Interface Ethernet2 detected 0 LLDP neighbors:
```

```
Interface Ethernet3 detected 0 LLDP neighbors:  
[-- output removed --]
```

```
Interface Ethernet33 detected 1 LLDP neighbors:
```

```
Neighbor 001c.b084.cfb0/"[ Arista Core ]", age 11 seconds  
Discovered 143 days, 5:44:44 ago; Last changed 143 days, 5:44:44 ago  
- Chassis ID type: MAC address (4)  
  Chassis ID      : 001c.b084.cfb0  
- Port ID type: Interface alias (1)  
  Port ID        : "[ Arista Core ]"  
- Time To Live: 120 seconds  
- Port Description: "GigabitEthernet1/0/48"  
- System Name: "3750-Core.gad.net"  
- System Description: "Cisco IOS Software, C3750 Software  
(C3750-ADVIPSERVICESK9-M),  
Version 12.2(37)SE, RELEASE SOFTWARE (fc2)  
Copyright (c) 1986-2007 by Cisco Systems, Inc.  
Compiled Thu 10-May-07 16:31 by antonino"  
- System Capabilities : Bridge, Router  
  Enabled Capabilities: Router  
- Management Address Subtype: IPv4  
  Management Address      : 192.168.1.70  
  Interface Number Subtype : ifIndex (2)  
  Interface Number        : 48  
  OID String              :  
- IEEE802.1 Port VLAN ID: 1  
- IEEE802.3 MAC/PHY Configuration/Status  
  Auto-negotiation      : Supported, Enabled  
    Asymmetric and Symmetric PAUSE  
    10BASE-T (half-duplex)  
    1000BASE-X (full-duplex)  
    1000BASE-T (half-duplex)  
    Other  
  Operational MAU Type   : 1000BASE-T (full-duplex)
```

```
[-- output removed --]
```

You can filter the output by specifying an interface, both with and without detail. Here is an example without detail:

```
Arista-7010T#sho lldp neighbors ethernet 33
```

```
Last table change time : 73 days, 2:51:53 ago
Number of table inserts : 5
Number of table deletes : 1
Number of table drops   : 0
Number of table age-outs : 0
Port      Neighbor Device ID      Neighbor Port ID      TTL
Et33      3750-Core.gad.net      [ Arista Core ]      120
```

Conclusion

That's it for now. Using LLDP will become second nature after a short time of using Arista gear. Although this chapter is quite short, when I cover Zero-Touch Provisioning ([Chapter 13](#)), understanding how LLDP works will help in your understanding of what I've done with some custom scripting there.

Chapter 4. Merchant Silicon

If you've shopped for data center switches with any of the major networking equipment vendors recently, you've likely heard the term *merchant silicon* thrown around. When I wrote the first edition of this book, there was a lot of back and forth between the major players about custom silicon versus merchant silicon, and which one is better.

Although many of the big players have adopted the ways of merchant silicon, there are still times when vendors will try to convince a potential customer that the Arista way is the wrong way. Let's take a look at the details and see whether one really is better than the other.

The Debate

To begin, let's define our terms:

Custom silicon

Custom silicon is a term used to describe chips, usually Application Specific Integrated Circuits (ASICs), that are custom designed and usually built by the company selling the switches in which they are used. Another term I might use would be *in house* when describing such chips. As an example, Cisco Nexus 7000 switches use Cisco-designed proprietary ASICs.

Merchant silicon

Merchant silicon is a term used to describe chips, usually ASICs, that are designed and made by an entity other than the company

selling the switches in which they are used. I might be tempted to say such switches use *off-the-shelf* ASICs, though that might imply that I could buy these chips from a retail store. I've looked, and Walmart doesn't carry them. As an example, Arista's 7280R switches use Broadcom's Jericho ASIC.

So that seems pretty cut and dried, but which one is better? That all depends on what you mean by *better*. Let's take a look at the benefits and drawbacks of each. First, these are the benefits and drawbacks of custom silicon:

Benefits of custom silicon

- Can be designed to integrate perfectly with a custom operating system
- Can be designed to support proprietary features
- Can be purpose built
- Can provide a significant competitive advantage due to the previous bullet points

Drawbacks of custom silicon

- Requires expensive on-staff expertise
- Requires expensive fabrication facilities
- Often slow to market
- Return on investment (ROI) can be slow
- Long ROI can lead to longer product life cycles

Now let's take a look at the benefits and drawbacks of merchant silicon:

Benefits of merchant silicon

- Easy to design around with well-supported APIs
- ASIC vendors are motivated to make stable, successful, and fast products
- Fast to market
- ASIC vendor does one thing: make ASICs
- No overhead involved (no expensive ASIC designers to staff, expensive manufacturing facilities to build and maintain, etc.)
- Easy to implement

Drawbacks of merchant silicon

- No custom or proprietary hardware features are possible (the chips might support proprietary features, but anyone that uses these chips has access to them)
- No inherent competitive advantage; any vendor can use same ASIC, although the implementation might be better with one vendor over another

Arista and Merchant Silicon

Arista uses merchant silicon exclusively for all of the reasons listed, but what about the drawbacks? The two drawbacks I listed for merchant silicon seem pretty severe to me, especially the one about there being no competitive advantage. I mean, isn't that why people buy one brand of switch over another, for the competitive advantages?

When I say there's no competitive advantage, I mean that there is no competitive advantage to using that ASIC compared to another vendor

using that ASIC. There are a couple of things to take into consideration with that statement, though. Let's take a look at the Arista 7280SR-48C6 as an example. It uses the Broadcom Jericho ASIC to deliver a 48 front-panel interface of 10 Gbps nonblocking goodness in addition to six additional 100 Gbps ports, all in a 1-rack unit (RU) box. Many other vendors offer similar switches that use the Broadcom Jericho ASIC. Arista's advantage in this space is that it has very efficient, modular, and portable hardware designs, and when a newer ASIC such as the Jericho2 came out, Arista quickly incorporated it into new products such as the 7280R2. Other vendors might very well have the same ability, so this advantage might be small or fleeting, but it exists nonetheless. Remember, too, that how a vendor implements an ASIC can have a tremendous advantage. This is one of the areas where Arista shines.

Another issue is the idea that no proprietary features are possible, and that's true so far as the ASIC hardware is concerned. Arista overcomes this limitation by differentiating itself with its Extensible Operating System (EOS). Much of this book is dedicated to the features of EOS, so I won't go into them here, but suffice it to say that EOS gives a significant competitive advantage to Arista that, so far as I've seen, can't be matched by any other vendor.

Proprietary features can be a good thing, but they can limit the ability to expand a network using different vendors and, in some cases, cause designs to be so tightly integrated into a single vendor as to cause severe limitations in the future. This limitation, commonly called *vendor lock*, can be a real problem when it comes time to upgrade the network. It's worth noting that Arista is generally against vendor lock

on the principle that given the choice, customers will choose the best option available. Instead of trying to force customers into buying a solution, Arista instead strives to be the best choice.

Perhaps the most compelling argument for the success of merchant silicon-based switches is that some of the biggest proponents of custom silicon have released merchant silicon switches, and why wouldn't they? If the SuperCool6000 ASIC is an advantage for Arista, and anyone can buy them from BroadTellSuperCom, every other vendor has every right to build the best switch they can, using the same hardware. It's up to you to decide whether Cisco's NX-OS is a better choice than Arista's EOS. At this point, I don't think I need to tell you what I think.

Fast-forward to 2019, and nearly all of the major networking vendors have followed Arista's lead and are now using merchant silicon. Arista's biggest competitor, Cisco, uses the Trident 2 chipset in its Nexus 9500s, though it does it in combination with its own custom ASICs.

Finally, there are some very cool aspects of Arista that aren't quite obvious to the casual observer. For example, though Arista uses the same ASICs as many other vendors, there have been times where it has improved the ASIC vendor's SDK in order to accomplish things with the chip that other vendors can't figure out how to do. I like to say that Arista has some of the best developers in the world, and that's what makes Arista the better choice, even when using the same off-the-shelf ASICs.

Arista Product ASICs

EOS offers the ability to show what ASIC is installed in your switch. To see the ASIC in use, use the `show platform ?` command. Here's the output from a 7010T, which shows that the ASIC is a *Trident*:

```
Arista-7010T(config)#sho platform ?
WORD          Forwarding Agent name
pending-state  Show all agents state
pkt           CPU packet info
schanaccel   S-Channel Accelerator Info
smbus         Smbus-device info
trident       Trident chip
```

Here's the output from a 7150S-24, which shows an *FM6000* ASIC:

```
Arista-7150(config)#sho platform ?
fm6000       FM6000 chip
pkt          CPU packet info
schanaccel  S-Channel Accelerator Info
```

The choices offered by each switch are different, depending on the ASIC installed. Here are the first few options for the 7010T:

```
Arista-7010T(config)#sho platform trident ?
CHIP          Chip name or pattern. eg. Linecard*
SLICE         Slice name or pattern
agent         Agent information
copp          Trident CoPP information
counters     Trident debug counters
[--output truncated--]
```

And here are the first few options presented on a 7150S-24:

```
Arista-7150(config)#sho platform fm6000 ?
acl           Alta ACL information
agileport    Agile ports let groups of 4 SFP+...
bst          Show internal bst registers
copp         Control plane policing
```

```
counters          FM6000 debug counters
[--output truncated--]
```

If you can get your hands on an Arista switch, I encourage you to dig around in these commands, because there is some really useful information in there. Although I used to include a list of what switches use what ASICs, I stopped doing that because they can update far more quickly than the production cycle of an O'Reilly book.

Certain ASICs provide certain features. For example, the DANZ feature called Agile Ports is available only on Arista 7150 switches due to the FM6000 ASICs they contain. Because the 7280R does not use the FM6000 ASIC, the switch does not support this feature. The 7280R switches use Jericho ASICs, and thanks to Arista's EOS and some of the world's best developers, these switches can support internet-scale routing tables. In fact, they can support multiple copies of said tables (see [Chapter 20](#)).

In your day-to-day network operation duties, do you care what ASICs are in your switches? Probably not. Still, it pays to know what you're talking about when the vendors come a-courtin'.

It's also important to consider what sort of power we're talking about here. Consider this: the Arista 7150S-52 supports 52 10 Gbps nonblocking Ethernet ports in a 1 RU switch, using one ASIC. The admittedly aging (but still widely deployed as of 2019) Cisco 6509 supports only 28 10 Gbps nonblocking (in theory) Ethernet ports, and that's in a full 15 RUs, consuming much more power and producing much more heat. It also uses a lot more than one ASIC to do it, which is one of the reasons that these big switches consume more power and

generate more heat. The 6509 is capable of many more enterprise features and is almost infinitely more expandable than the Arista 7150S-52, so it's not a strictly apples-to-apples comparison—unless all you need is nonblocking 10 Gbps port density, in which case the Arista 1RU switch wins handily.

I should note that not all Arista switches use only one ASIC. The 7280R2 supports many 10 Gbps ports in a 1 RU chassis, but it does that using Jericho+ ASICs, which happen to be the same ASICs used in the Arista 7500R2 chassis switch.

The Cisco 6509 is a great switch, and I'm not knocking it here. It is, however, a great example of the long product cycle induced by the custom silicon mindset. Though it supports high-density 10 Gbps blades, with only 40 Gbps available in each slot (using Sup 720s), those blades are highly oversubscribed. It's been around for a long time, but there are still a lot of them out there. Newer switches from competing vendors have better specs, but most of them are using merchant silicon these days.

There is one more potential benefit to merchant silicon, and that is the possible future of Software Defined Networks (SDNs). Think of an SDN as a cluster of switches, all controlled by a single software brain that is running outside of the physical switches. With such a design, the switches become nothing more than ASICs in a box that receive instructions from the master controller. In such an environment, the switch's operating system would be much simpler, and the hardware would need to be commoditized so that any vendor's switch could be added to the master controller with ease. Merchant silicon-based

switches lend themselves to this type of design paradigm, whereas a custom silicon solution would likely support a master controller from only that switch's vendor.

NOTE

This is a bit of an oversimplification of the idea behind SDNs, but it does seem to excite the executives who hear about it. We're a few years away from this application in my opinion. Currently, SDN-type features are being used for things like security and monitoring.

To add to my statement that we're "still a few years away" from such an application, while writing the second edition, I was amused as I read that some three years after the first edition was published because I still think we're "a few years away" from the SDN dream. There's an old joke from the telecom world that ISDN stands for I Still Don't Know. I find it amusing to be able to recycle old telecom jokes.

Will SDN become a widespread reality? I don't know. I think the idea has merit, but as of mid-2012, er, 2019, I wouldn't be making any purchasing decisions based on it. That viewpoint may change in the next few years. Arista has a cool new tool called CloudVision that is a step in the right direction, and we talk about that a bit later in the book.

There's another topic that seems to come up when merchant silicon is discussed, and that's the idea of white-box switches. The concept of white-box switches is simply that you can buy a switch as a generic commodity product that will accept whatever network operating system you'd like to install. Although this sounds like a good idea to many, in my experience very few installations are actually following through with it. The consultant part of my brain seems to think that executives would rather have a company they can yell at when things

go wrong (the *one throat to choke* paradigm). Additionally, if you want the best network operating system, I'm going to go ahead and recommend EOS, and if you have EOS, you might as well get some of that sweet Arista hardware to go with it, no?

NOTE

Here's where I begin to wonder whether I'm in too deep and have become biased. Then I think about the hundreds of companies I've worked for or with and think about what I know about the industry as a whole. When I step away from my Arista goggles for a bit and survey the landscape, I realize why I'm still at Arista some six years later: Arista still has the best solution.

Conclusion

So, which is better: custom silicon or merchant silicon? As far as Arista is concerned, merchant silicon is the path they've chosen. I can't tell you which is better, because I've seen great products from both camps. I will tell you that I really like what I'm seeing as a result of the competition caused by vendors moving to merchant silicon. Take EOS, for example. I think it's the best networking operating system I've ever seen. If that came to be as a result of using merchant silicon, then I'm a fan.

Chapter 5. Fabric Speed

One of the things you'll hear over and over from switch vendors is how fast their switch is, how fast the fabric is, or how much backplane capacity it has. But what does all that mean?

NOTE

This chapter is not intended to explain every possible detail of switch fabrics, but rather to help you understand what the term *fabric speed* means in a general sense. Entire books could be written on this topic, but my goal here is to help you understand the confusing and often misleading numbers that many vendors include in their switch specification sheets.

In a top-of-rack (ToR) switch, the fabric is the interconnections between all the interfaces. The term *backplane* in this case is pretty much synonymous with *fabric* (though probably inaccurate). On a chassis switch, the terms can be thought of a bit differently.

On a chassis switch, each module can have a fabric, and interfaces within a module can switch between each other while staying local to the blade. When a packet sourced on one blade must travel to another blade, though, the packet needs a path between the blades. The connections between the blades are often called the backplane, although really, that term is more about the hardware connecting the modules to one another. Semantics aside, what you need to know is that there is a master fabric connecting all these modules together in a

chassis switch.

In modern chassis switches, like the Arista 7500R and Cisco Nexus chassis models, the backplane fabric resides in hot swappable modules in the back of the chassis. In older switches, like the Cisco 6509, they used to consume slots in the front of the switch (at the expense of other useful modules). Later, Cisco incorporated the fabric modules into the supervisors on the 6500s so that more slots could be used in the front for interfaces.

There are many ways in which modules can connect to the backplane fabric. There might be Application-Specific Integrated Circuits (ASICs) on the modules that allow packets to flow intramodule without intervention from the fabric modules. Or, the fabric modules might interconnect every port in every module. How the modules are constructed and how they connect to the backplane both determine how fast the backplane fabric is. Really, though, many vendors play games with numbers and try to hide what's really going on. Let's take a look at why.

Let's look at our simple three-port 10 Gbps switch again, as shown in Figure 5-1. In this switch, the fabric is simple and nonblocking.

Nonblocking means that each port is capable of sending and receiving traffic at wire speed (the maximum speed of the interface) to and from any other port. A nonblocking switch is capable of doing this on all ports at once. Remember, on these fabric drawings, that the ingress is on the left and the egress is on the top. This means that when you see interface e1 in two places, it's the same interface. It exists in two places to show how packets come in and leave through the fabric.

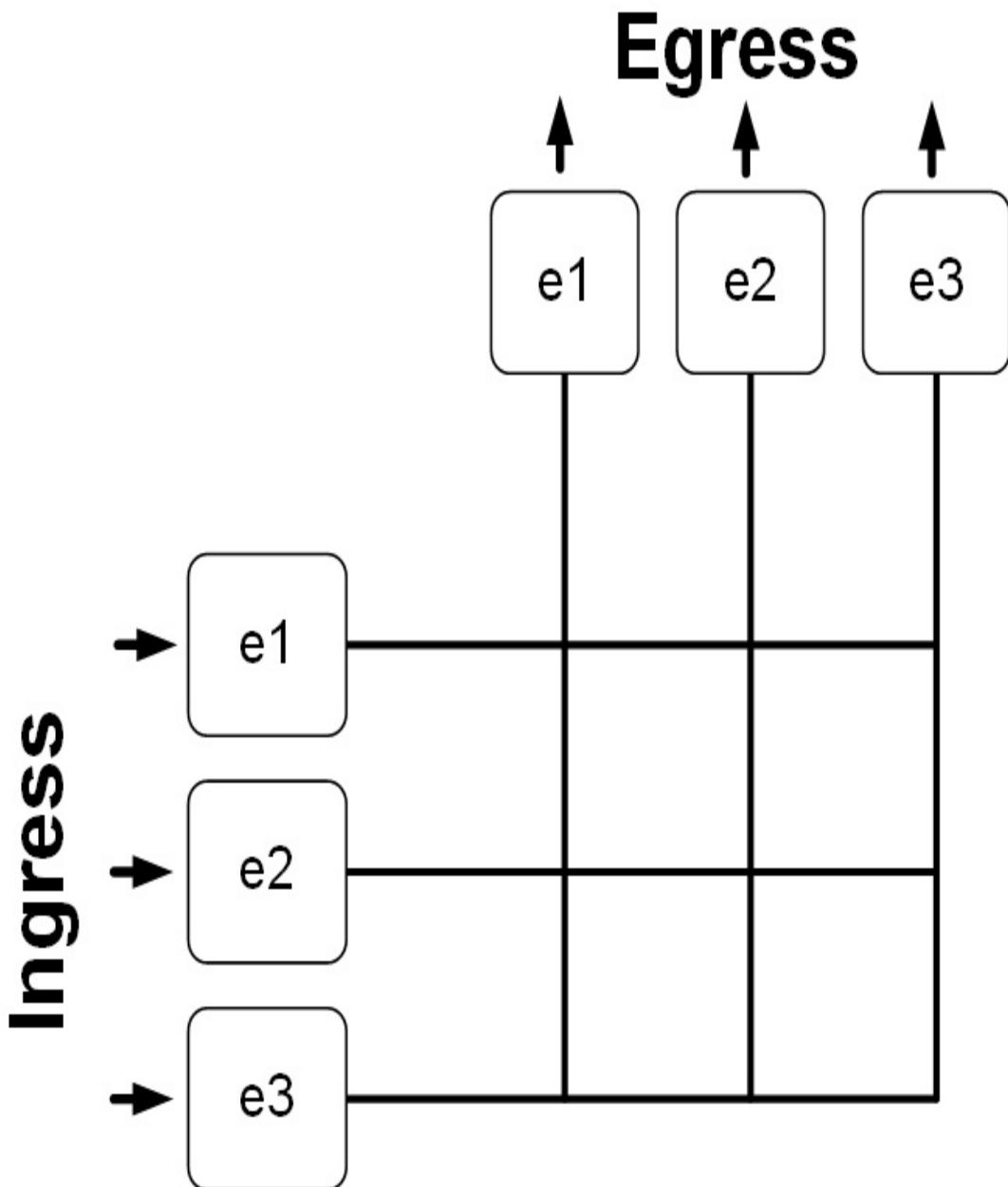


Figure 5-1. Simple three-port switch fabric

Now imagine that our simple switch used ASICs to control the flow of packets between ports. There is one ASIC on the ingress and one on the egress. The kicker here, though, is that each ASIC is capable of forwarding only 10 Gbps at a time. Figure 5-2 shows this design.

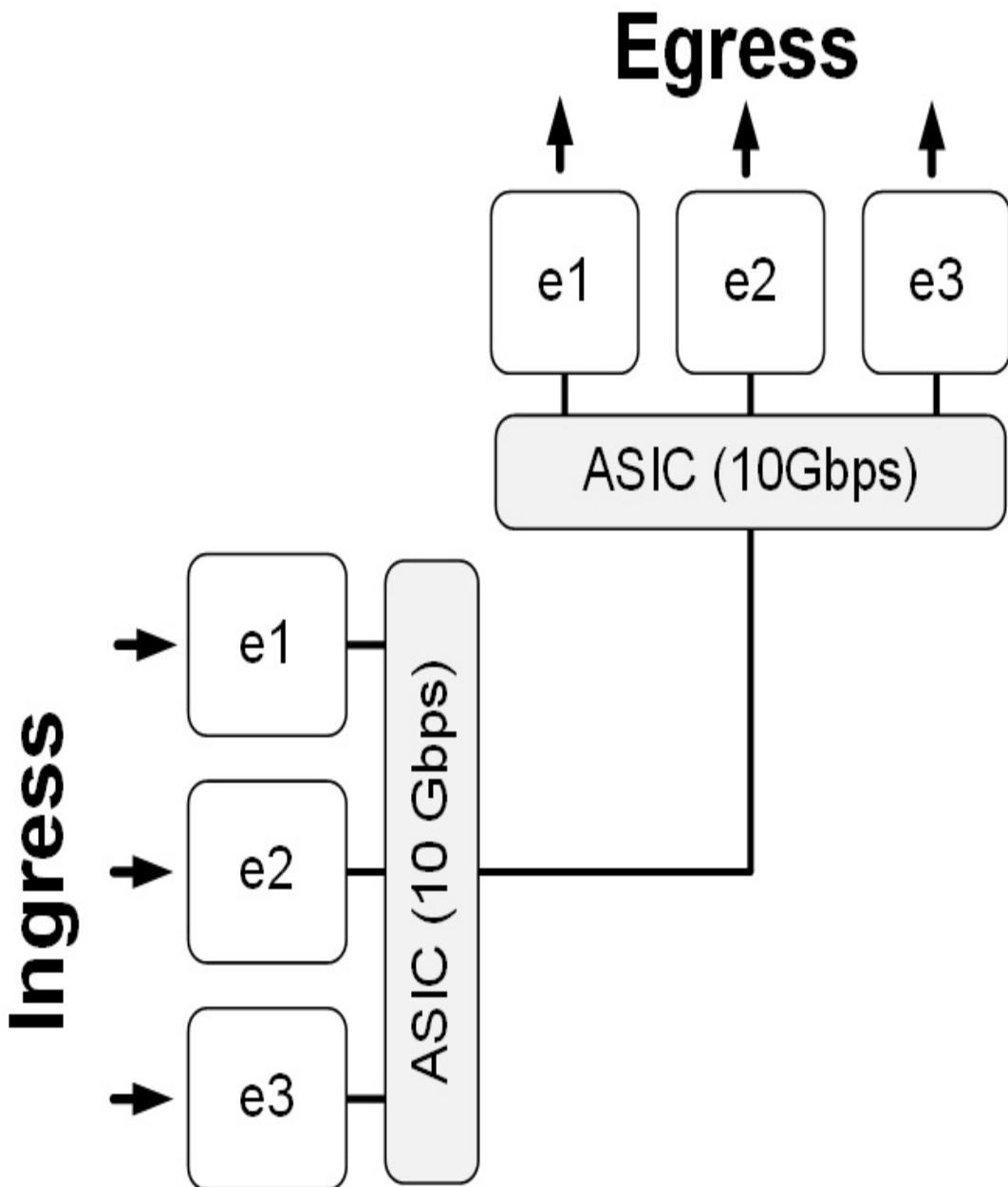


Figure 5-2. Simple three-port switch fabric with ASICS

All of a sudden, with the addition of the 10 Gbps ASICS, our switch has lost its nonblocking status (assuming that each port is 10 Gbps). Though each interface forwards bits to its connected device at 10 Gbps, should more than two ports be used at the same time, their maximum combined transmit and receive can be only 10 Gbps. Think that sounds

bad? Well, it is, assuming that you need a nonblocking switch. The truth is, very few networks outside of the data center require real nonblocking architectures. Most eight-port gigabit switches found in the home are blocking architectures just like this. But how often do you really need to push 100% bandwidth through every port in your small office/home office (SOHO) switch? Probably never. And guess what? Building oversubscribed switches this way is inexpensive, which is a plus in the SOHO switch market.

FABRIC SPEED MARKETING

When I first started learning about Arista switches, it was during a technology bake-off for one of my clients. Arista had told me that its fabric was nonblocking, so I decided to document what all of the other vendors were doing in their switches only to discover a pile of misinformation ranging from simple omission to outright misdirection.

The biggest offense I found over and over was advertising a fabric that was the equal of all of the ports on the switch. Why is that a problem? Because Ethernet today is full duplex. If you have 10 ports of 10 Gbps, you don't need a 100 Gbps fabric to be truly nonblocking—you need a 200 Gbps fabric, and that's at a minimum! A 100 Gbps fabric would be 2:1 oversubscribed.

Even today I often ask students in my classes how much backplane capacity would be needed for a 10-port 10 Gbps switch and most people will respond, "100 Gig." As outlined in the chapter on Arista Products (Chapter 6), Arista does not sell over-subscribed switches.

Let's look at a bigger switch. [Figure 5-3](#) depicts an eight-port 10 Gbps switch. This switch is nonblocking, as evidenced by the fact that each interface has a possible full-speed connection to any other interface.

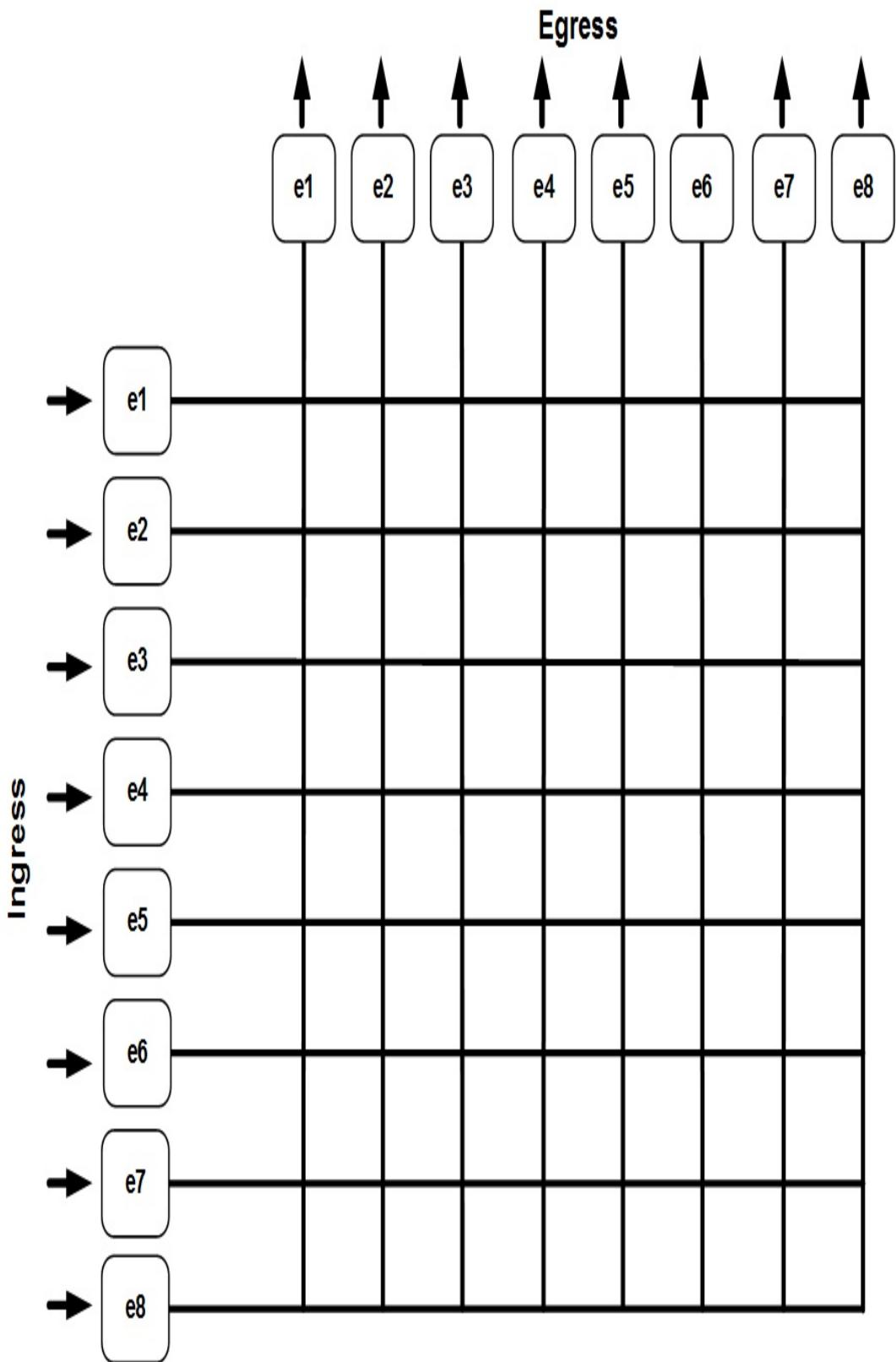


Figure 5-3. An eight-port 10 Gbps nonblocking switch

How might this switch look if we used oversubscription to lower costs?
If we used the same 10 Gbps ASICs, each controlling four 10 Gbps
interfaces, it might look like the switch in [Figure 5-4](#).

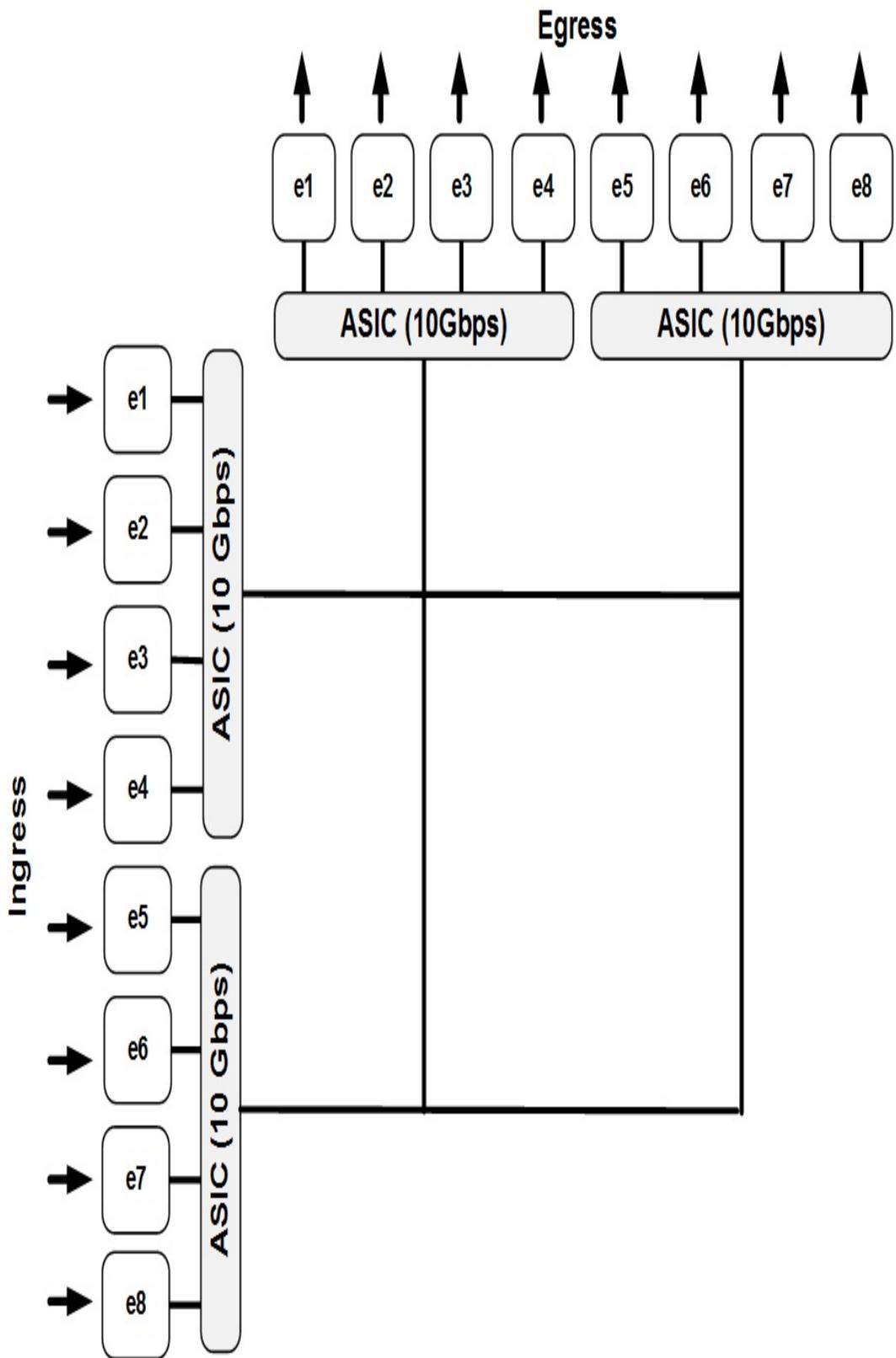


Figure 5-4. An oversubscribed eight-port 10 Gbps switch

If you think stuff like this couldn't happen in a modern switch, think again. The Netgear SOHO GS108T switch advertised that it had "eight 10/100/1000 Mbps ports, capable of powering 2000 Mbps of data throughput." Since the first edition of Arista Warrior, this switch has been updated to the GS108Tv2, which now advertises "16 Gbps full duplex" of bandwidth, so even the SOHO market is getting into the nonblocking game!

Even the big iron is commonly oversubscribed. Though a bit dated now, the Cisco Nexus N7K-M132XP-12L module, shown in [Figure 5-5](#), sported 32 10 Gbps ports, but each group of four ports supported an aggregate bandwidth of only 10 Gbps. This is illustrated by the fact that one out of each of the four ports could be placed into dedicated mode, in which that port was guaranteed the full 10 Gbps.

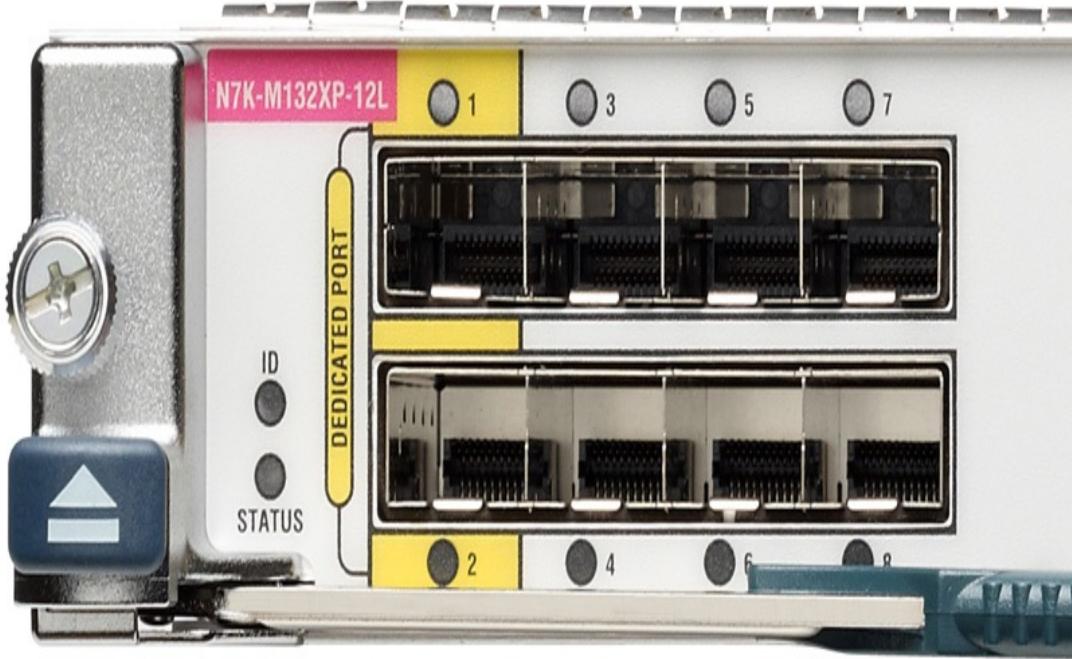


Figure 5-5. Cisco M7K-M132XP-12L module with dedicated ports highlighted

This is not a knock on Netgear or Cisco. In fact, in this case I applaud them for being up front about the product's capabilities and, in the case of the Cisco blade, for providing the ability to dedicate ports. On a server connection, I'm perfectly OK oversubscribing a 10 Gbps port because most servers are incapable of sending 10 Gbps anyway. On an inter-switch link, though, I'd like to be able to dedicate a 10 Gbps port. My point is that this oversubscription is very common, even in high-dollar data center switches.

AMUSING DIGRESSION

As the data center networking wars have heated up, oversubscribed ports are less and less common, and I'd like to think that's because Arista led the way with massive switches like the 7500, the 7500E, and now the 7500R that supports a ridiculous number of high-speed ports without oversubscription.

To be fair, my earlier example of the Cisco blade is not as relevant today because that blade is a few years old, but I'll say this: ask any Nexus 7000 owner what it took to get from that oversubscribed switch architecture to a nonblocking one and you might just find a customer who migrated to Arista.

What if we could get a single super-ASIC that could control all of the ports? Such a switch might look like the one illustrated in [Figure 5-6](#). In this switch, each port still has full 10 Gbps connectivity to any other port on the switch, but there is no oversubscription. This super-ASIC can handle the high-bandwidth requirements of a non-blocking 10 Gbps switch.

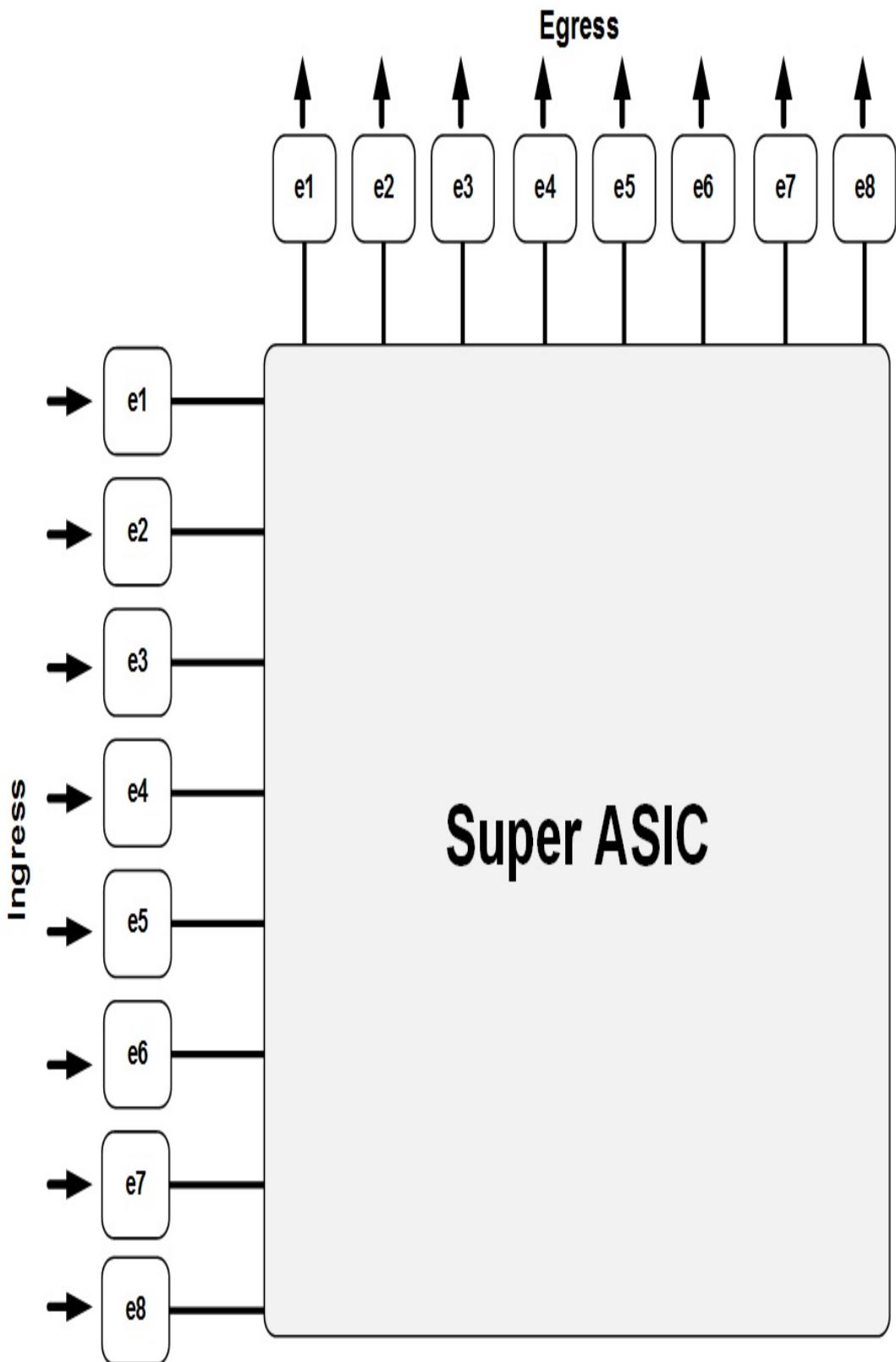


Figure 5-6. Nonblocking eight-port ASIC-based 10 Gbps switch

Some switch vendors take a different approach and cascade ASICs. Take a look at [Figure 5-7](#). Here, there are 16 ports, divided into two modules. Each of the four ports is controlled by a single 10 Gbps ASIC. Each of those ASICs connects to a 40 Gbps ASIC that manages interconnectivity between all of the modules. This works, but it doesn't scale well. I've seen this type of solution in 1-rack unit (RU) or 2 RU switches with only a couple of expansion modules. Note that the backplane speed cannot be improved with such a design, because it is purposely built for this scale.

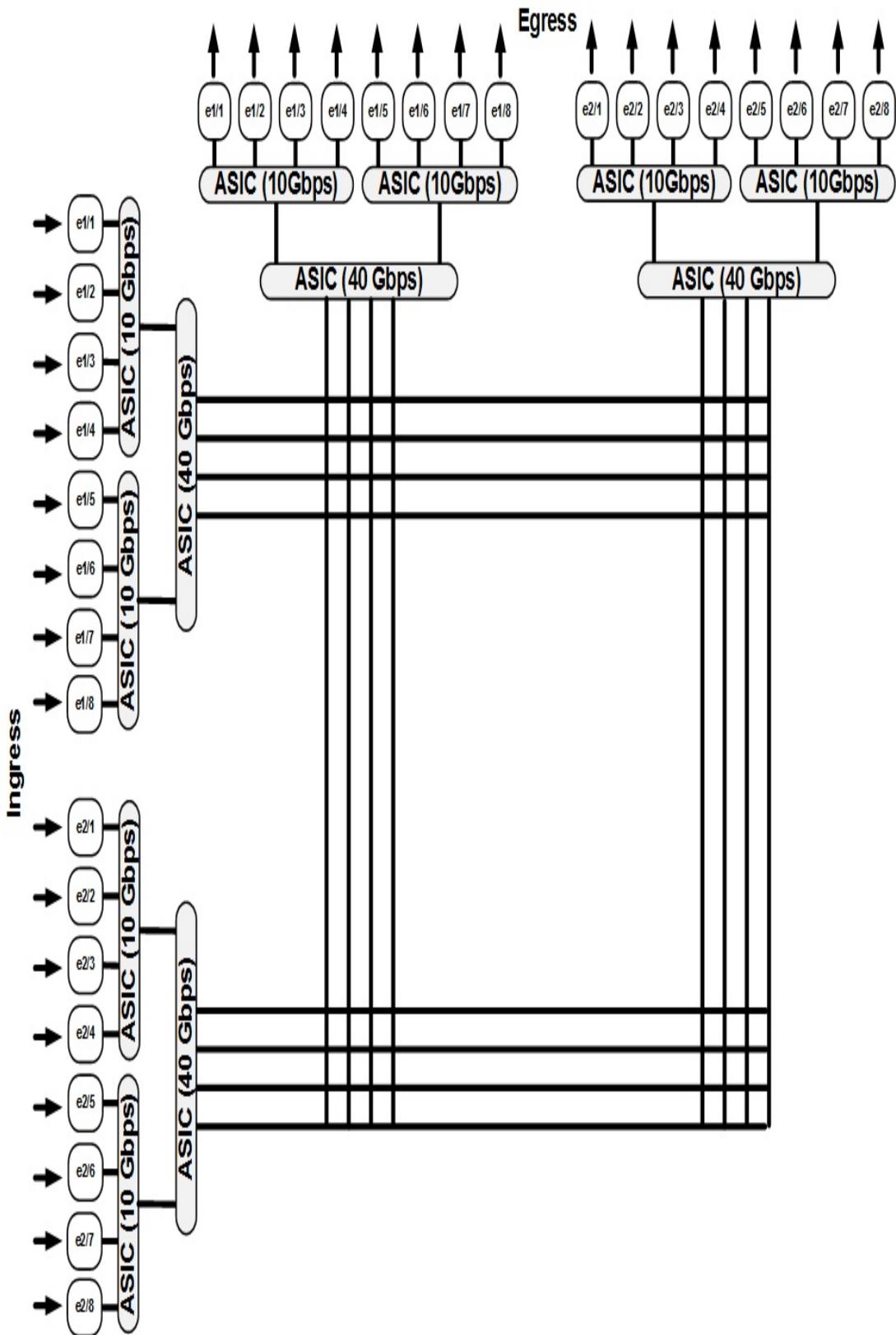


Figure 5-7. Cascading ASICs

This type of switch will get you 10 Gbps connectivity, but at a lower aggregate throughput, which equates to oversubscription. That's fine for an office switch, and for some data center switches, but for high-end networking, I'd like to see a nonblocking eight-port 10 Gbps switch. That would look something like the switch depicted in [Figure 5-8](#).

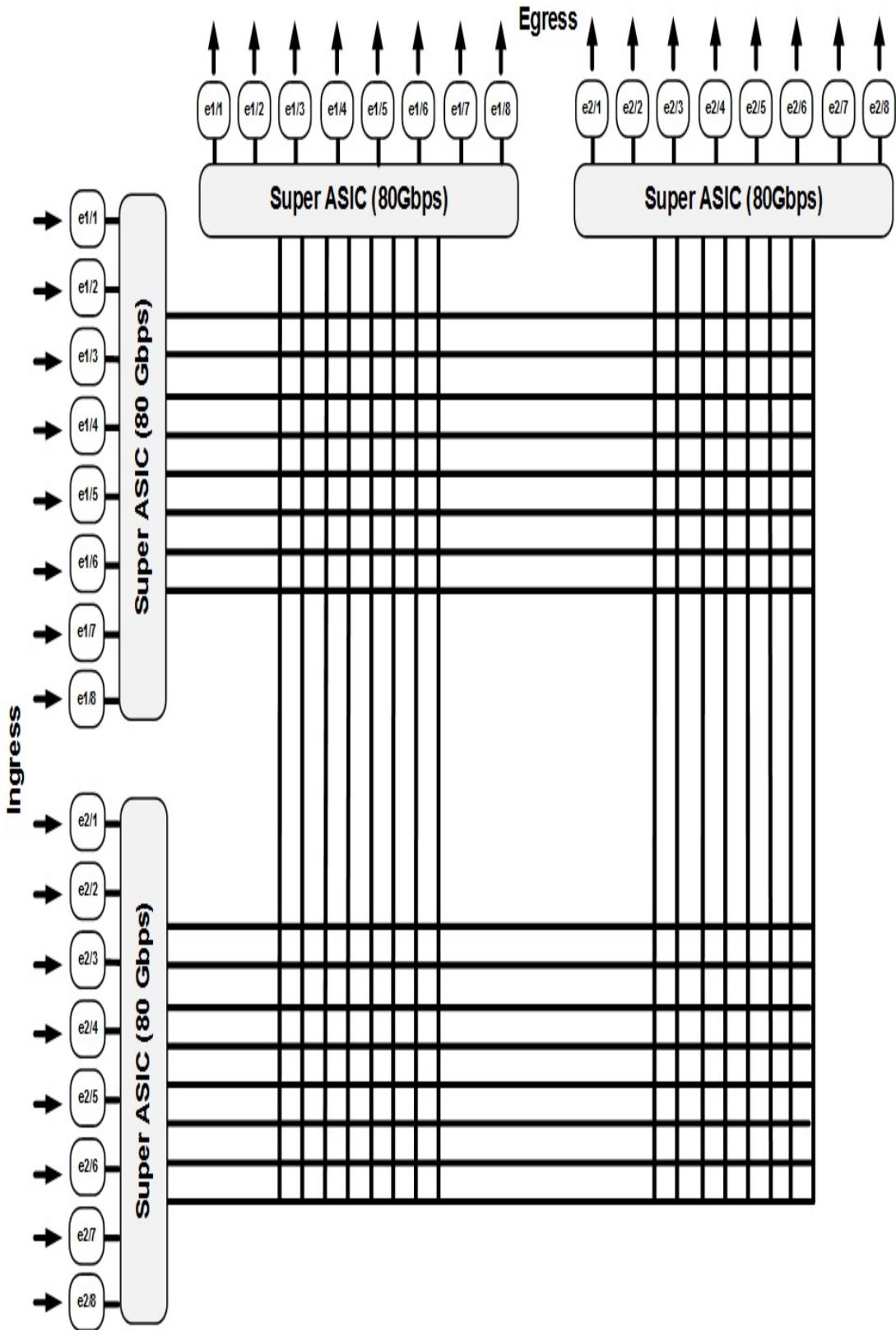


Figure 5-8. Nonblocking 16-port 10 Gbps modular switch

This is better, but it's still not scalable because we can't add any connections to the core fabric. So, what if we used even bigger, better ASICs to mesh all of our modules together? In [Figure 5-9](#), ASICs perform the intermodule connectivity. In this model, assuming that the hardware was designed for it, we could theoretically add more modules, provided the ASICs in the center could support the aggregate bandwidth.

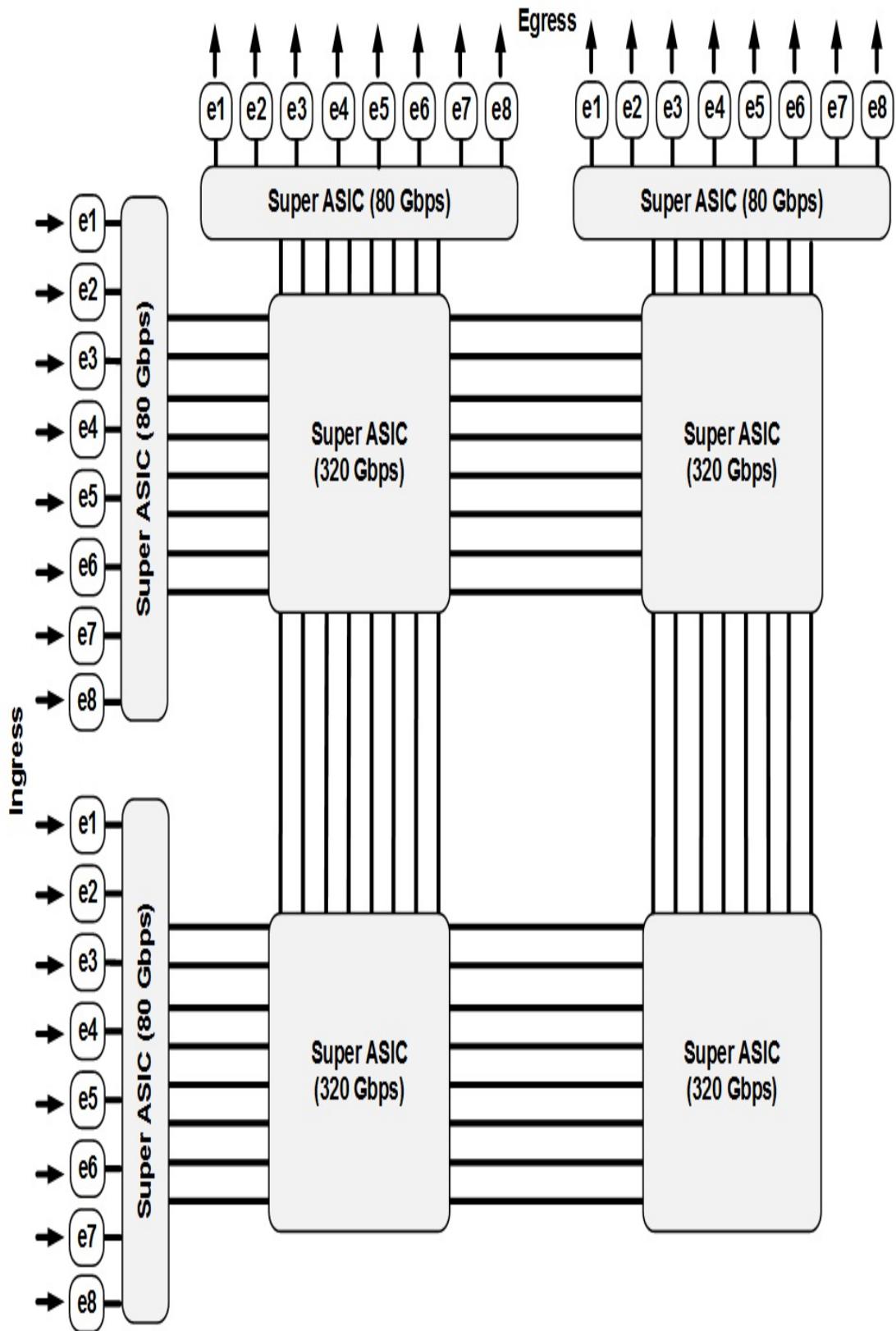


Figure 5-9. Nonblocking ASIC-based 16-port modular switch

When it really comes down to it, how the backplane functions rarely matters to the end user. What matters is whether the switch is truly nonblocking. If that's done with one ASIC or 12, does it really matter? Probably not to those of us who are designing and building networks, but to the guys who write the code, it can make their lives easier or more difficult.

Arista switches incorporate a few different designs, depending on the design requirements of the switch. The Arista 7050S-64 (a 1 RU switch) uses a single Broadcom Trident+ ASIC to switch 64 nonblocking 10 Gbps ports. To put that in perspective, that's more nonblocking port density than a Cisco 6509 loaded with the latest supervisors and seven eight-port 10 Gbps modules. There are a lot of ASICs in a fully loaded 6509.

That's not to say that the single ASIC approach is always the right solution. The Arista 7500E chassis 48-port 10 Gbps nonblocking modules has one ASIC per every eight ports (six per module). Each module has a total of 1.25 Tb access to the backplane fabric (648 Gbps TX and 648 Gbps RX), which translates to the possibility of each slot supporting multiple 40 Gbps and even 100 Gbps interfaces, all of which would be nonblocking. Note that in this design there can be a slight latency penalty when frames are forwarded between ASICs, though this type of penalty is usually a problem only in extremely low-latency environments like high-frequency trading.

Conclusion

The fabric or backplane is the heart of a networking switch, and Arista

once again led the industry in providing nonblocking *undersubscribed* switches in a world where most everyone else was *oversubscribing* their fabrics and delivering products that couldn't match the performance of a simple nonblocking architecture.

Chapter 6. Arista Products

The Arista product portfolio can seem a little overwhelming at first because many of the switches look similar and there are some features and software products that have names that don't make their function immediately obvious, but after you dig in, you'll realize why each product exists and how you might use it in your own networks. Since the first edition of *Arista Warrior* was published, Arista has expanded with more models and a variety of software-only solutions that I cover here, as well.

Switches

Arista now has so many switches that the marketing material has changed because putting them all onto a single image would mean that each switch would be too small to see well. In the first edition of this book, I put in a picture and a brief summary of each current model, but I've moved away from that approach in this edition.

AMUSING DIGRESSION

As a fun aside, when the first edition of this book was ready, Arista invited me to be a guest speaker at one of its Customer Exchange events in Santa Clara, California. Remember, I was not yet an employee, so I was pretty excited to learn that I would be following none other than Silicon Valley visionary Andy Bechtolsheim. After I accidentally stepped on his toes (yes, really), he got up to speak, where he proceeded to announce a new set of switches to raucous applause. When he was done, I got up and had to explain how my cool new book, which was to hit shelves in mere days, was just made obsolete by Andy. Thankfully, I could turn that into some self-deprecating humor, but that day I learned never to follow Andy during public speaking engagements.

Up until 2018 or so, Arista switches were all purpose built for the data center, and you probably wouldn't have bought an Arista switch for a VoIP deployment that required Power Over Ethernet (PoE). Though you could certainly use an Arista switch in your enterprise office environment, the company didn't make a PoE model, the switches weren't (and still aren't) stackable, and Arista didn't really chase the office-switch market. That changed in 2018 when Arista purchased the WiFi company Mojo Networks, which has allowed Arista to sell PoE switches along with wireless devices. Arista has entered the enterprise space with gusto, so I have to imagine that PoE switches will become more common in the Arista product line along with other enterprise-class devices.

The current marketing material displays the image shown in Figure 6-1 or a derivation of it.



Figure 6-1. The Arista switch portfolio

I'm not a huge fan of this image for various reasons, but the fact remains that there are so many switch models now that this is probably about the easiest way to convey the first step in choosing a switch for most customers. Instead of going over every individual model only to have that information be out of date more quickly than I can write another edition, in this book I'm going to focus on features that make Arista switches great while pointing out some model-specific things that I think you'd likely want to know about.

The way I tell people to think about which model switch they need is to first consider port density and line speed. If you need 100 Gbps interfaces, that will limit your choices, and looking at other models is a waste of time. Additionally, if you need 1,500 interfaces in a single box, you're probably going to be looking at chassis switches. Next, I tell people to consider what features they need. Because many features are hardware based and thus Application-Specific Integrated Circuit (ASIC) specific, whittling down the choice based on feature set then makes sense. Following these steps (in either order, honestly) will generally get you to within a few product families after which your account rep and customer engineer can help you pinpoint the best switch or switches for your requirements.

For example, if you need to support full internet routing tables, you need a device with an R suffix such as the 7280R or 7500R (or the even newer 7280R2/7500R2). If you need to have 400 40 Gbps interfaces in one box, you're looking at a 7500R. Now there are options like the 7500R2, and it's time to get into detail with your sales team to figure out what the best choice is for your network.

Arista switches are currently divided into two types of hardware layout —one and two rack-unit (RU) top-of-rack (ToR) fixed configuration switches, and multirack-unit chassis-based switches. There are multiple hardware layouts within each type, but I’m not going to that level of detail here. You can find information regarding the current stable of Arista products at [the company’s website](#). The information contained within this chapter is accurate as of final editing of this book. Technologies change quickly, so contact your Arista sales representative for the most current information. For a quick comparison of all the current Arista switches, the *Arista Product Quick Reference Guide* is an invaluable tool.

Instead of looking at specific products, I’m going to cover some of the common features or bits of information that I think you’ll find helpful when navigating the world of Arista products:

- Switch names
- Power
- Airflow
- USB
- Rails
- High-speed Ethernet
- Nonblocking architecture
- FlexRoute
- AlgoMatch
- Optics

- Software products

Let's begin with how the Arista switch model names work.

Switch Names

It's tempting to assume that a higher model number is "better" than a smaller model number, but that's not always true. For example, the 7160 models might seem like a newer version of the 7150, but the 7150 still does things that the 7160 cannot do.

Looking at a model number, there are some things that can be derived from it. The switch DCS-7050TX-96-F tells me that this is a Data Center Switch (DCS) model 7050 with BaseT connections (RJ45 copper) that are 10 Gbps (X) and that there are 96 10 Gbps interfaces possible (48 front-panel RJ45 interfaces and four 10/25/40/100 interfaces that can be split into an additional 48 10 Gbps interfaces). Finally, the "-F" at the end means that the switch has front-to-back airflow fans. Switches that have "S" in the name (7150S) are Small Form-Factor Pluggable (SFP)-based, whereas Q switches like the 7050QX are Quad Small Form-Factor Pluggable (QSFP)-based. Models with "C" in the name (7280SR-48C6) have 100 Gbps ports (in this case, six). "-M" indicates extended memory. Let's look at a nice example:

DCS-7280SR-48C6-M-F

DCS: Data Center Switch

7280SR: Model 7280 with SFPs and enhanced routing capability

48C6: 48 ports (SFPs) with six 100 Gbps ports

M: Enhanced memory (32 GB in this case)

F: Front-to-back fans

On some switches you might see things like “-SSD” for an added solid-state drive (SSD) drive, or “-CL” for onboard high-accuracy clocks, though that tends to be on older switches like the 7150s.

Power

In the first edition, I wrote that “all Arista switches support multiple, redundant power supplies as well as multiple fans, all of which are hot swappable.” Today, that is mostly true. All switches support multiple power supplies, but the Arista 7010T and 7020T do not have hot-swappable power supplies. Those models also have only one fan module, though it is hot swappable and reversible. All of the top-of-rack switches have the fans and power supplies on the back of the switch, whereas some of the chassis-based switches have the power connectors on the front. Heck, the larger 7300 chassis switches actually have power supplies on the front and the back because the chassis are so large and need more power supplies than can be crammed into the back. [Figure 6-2](#) shows an older Arista 7050S-64 switch with the power supplies and fans removed. Though the model is a bit dated by today’s standards, it still works as an example because the basic architecture of the 1 RU chassis has remained relatively unchanged.

NOTE

The side of the switch with the network interfaces is the front.

I was pleasantly surprised when I unpacked my first Arista switch to see that the power cables supplied included C13-C14 connectors. These connectors are commonly used in data centers that employ large power distribution units (PDUs) within racks. Normally, I end up with a box of unused power cables that are replaced with the C13-C14 variety. Arista once again shows that these switches are purpose built for the data center.



Figure 6-2. Arista 7050S-64 switch with the power supplies and fans removed

Figure 6-3 shows a comparison between an older 7150S power supply and a newer 7280R power supply. I point this difference out because although they look similar and will fit in the same slot (but won't actually install), the power supply on the 7280 has a diagonal handle to indicate that it is a higher-wattage supply than the one from the older 7150. The reasoning for this is that the older power supply cannot be used in the newer 7280R, so having a quick cosmetic and physical indication that they're different is a good thing.



Figure 6-3. Old standard wattage (bottom) versus newer (top)

Airflow

As of this writing, airflow is configurable in almost all Arista switches with the exception of the 7500s and the 7280CR-48. All red-handled power supplies and fans pull air from the front (interface side) of the switch back through to the back. All blue-handled power supplies and fans pull air from the back (power side) of the switch to the front.

Think of it this way: if you see a red fan, you'll feel hot air. If you see a blue fan, you'll feel cold air being sucked in. Figure 6-4 shows two switches, each with different airflow options installed.



Figure 6-4. Color-coded fan modules indicating airflow direction with blue (top) denoting back-to-front airflow and red (bottom) denoting front-to-back airflow

USB

All Arista switches have at least one usable USB port, and by *usable*, I mean if you slap a USB stick in there, EOS will mount it as the device `USB1`: After decades of working with other vendors' switches that had USB ports that you couldn't actually use, this is a huge deal to me. Not only that, but they supply power, too! I've actually run a Raspberry Pi powered from the USB port of an Arista switch. [Figure 6-5](#) shows the front of a 7150S-52 switch showing the console, management, and USB ports on the front.

One of the things that I like to point out to people is that this USB port is not limited to just sticks. Want to add an SSD but didn't order a switch with one? Get a USB SSD drive and plug it in, and the switch will mount it! Just make sure that it's not formatted with NT File System (NTFS) as that's a licensed file system that's not supported natively by Linux. ExFAT is not supported for the same reason. FAT32 works just fine, though.

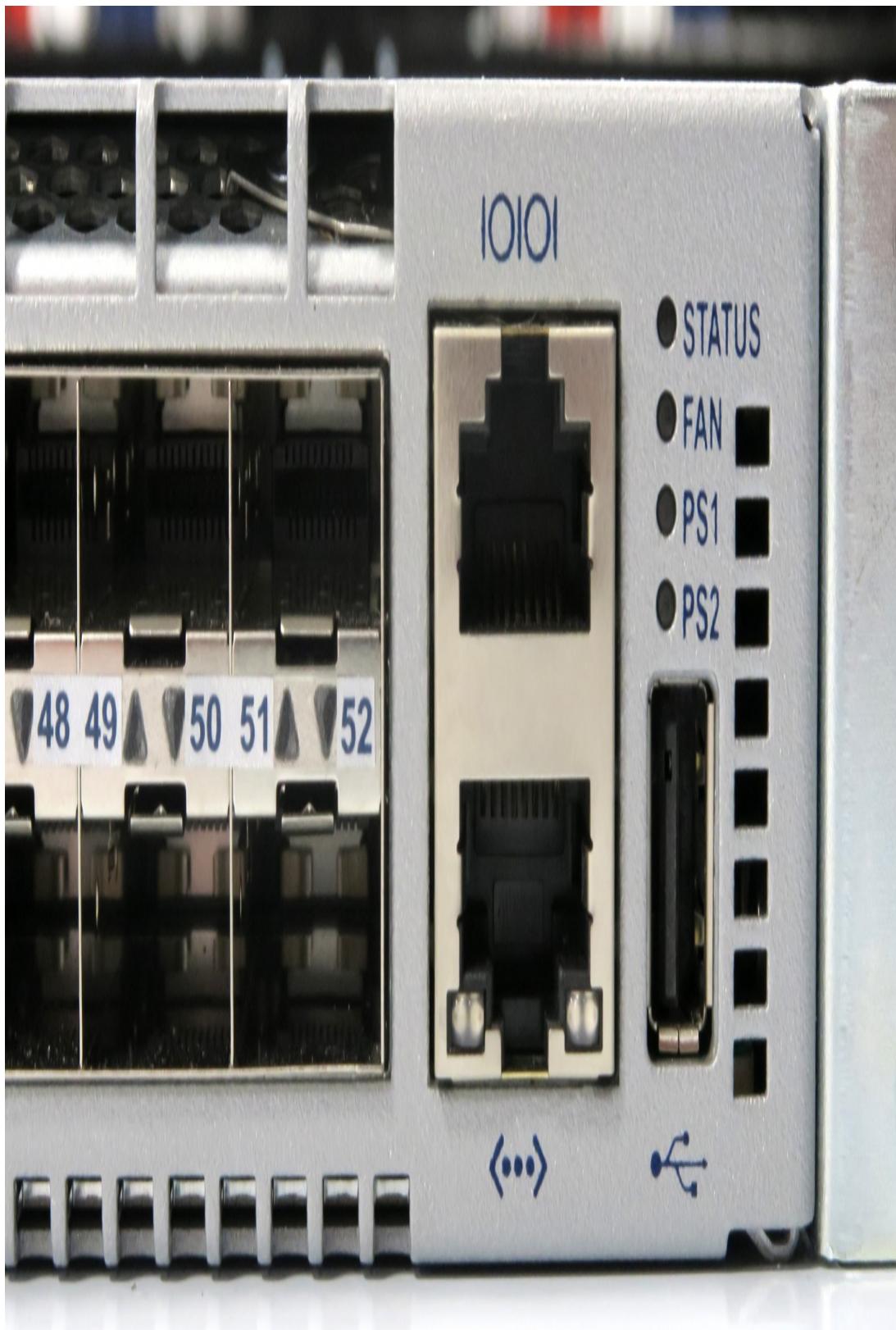


Figure 6-5. USB, management, and console ports on the front of a 7150S-52 switch

The Arista documentation for EOS 4.21.1F says:

The flash drives must be formatted with the FAT or VFAT filesystems. Windows NT File System (NTFS) is not supported.

VFAT? That's some Windows-95 tech, isn't it? VFAT stands for *Virtual File Allocation Table* and was first introduced in Windows 95 as a way to support long filenames on FAT file systems. VFAT works with FAT (which is FAT12 if you want to be picky), FAT16, and FAT32, but because no one uses any form of FAT other than FAT32 with extended filenames today, the Linux tools such as `mkfs.vfat` just reference the entire mess as VFAT.

I took a brand new 64 GB USB stick and formatted it on my Mac with the FAT filesystem (the FAT mentioned in the OSX Disk Utility is actually FAT32). Here's what my 7010T switch saw when I inserted the drive and poked around:

```
7010T#dir usb1:  
Directory of usb1:/  
  
61831151616 bytes total (61828399104 bytes free)
```

Here's what Linux thinks of my FAT32-formatted USB stick:

```
7010T#bash mount | grep usb  
/dev/sda1 on /mnt/usb1 type vfat  
(rw,dirsync,noatime,gid=88,fmask=0007,dmask=0007,allow_utime=0020,  
codepage=cp437,iocharset=ascii,shortname=mixed,flush,errors=remount-ro)
```

The switch sees FAT32 as VFAT. What matters, though, is that it works! How about a nice 1 TB drive?

```
7010T#dir usb1:
```

```
Directory of usb1:/  
999616839680 bytes total (999614021632 bytes free)
```

Nice! I would probably use ext4 for any drive connected to a Linux box, but then you wouldn't be able to read it natively in Windows or on a Mac, though there are add-ons for both that will support it. Although other filesystem formats like NTFS, ext4, or the macOS Extended filesystem are better options for various reasons, FAT32 is the best choice if you want the drive to be read on all of them.

I used to love to point out that all Arista switches had all of the management and console ports on the front, but because Arista has been able to cram more and more high-speed interfaces onto 1 RU chassis switches, and more front-panel interfaces trumps convenience any day, some switches now ship with the management, console, and USB ports on the back of the switch. Figure 6-6 shows the back of a 7280R, where the console and management ports reside. If you look closely, you can see that there is a USB port between them, and it's actually labeled USB2 because there's another one on the front of the switch, which is USB1.



Figure 6-6. Console, management and USB2 ports on the back of a 7280R switch

Rails

Although different Arista switches might have different rails, I feel the need to point out the very cool rails that are now shipping with some of the newer models like the 7280Rs. I'll get right to the point: someone in the networking world finally looked at what the server vendors have been doing for decades, and the new rails require no screws whatsoever to be mounted in a rack. Huzzah! [Figure 6-7](#) shows a 7280R with the rail sitting next to it. Looking at the side of the switch, you can see a series of small protuberances that align with the holes on the rails. You simply take the rail, line up the holes, and slide it to the right a bit until it clicks, and just like that, you've mounted the rail on the switch. No more tiny screws that I end up dropping into the air vents in the data center!



Figure 6-7. Arista 7280R switch with one of its mounting rails

But what about the part of the rail that mounts to the rack? That's even better! The mating rail for the cabinet simply clicks into the square holes like countless server rails do. There are no tools, no little rack nuts, no bleeding fingers, no lost parts, and no profanity. It's heaven.

Figure 6-8 shows the rack-connecting end of the 7280R rail kit, and true to form, these same rails are used on other Arista switches that are the same form factor and general design, meaning that you don't need to have different rails for every damn switch. Brilliant!

As someone who has probably spent a combined year of my life sitting in data centers attaching rails to both switches and cabinets along with the requisite blood, sweat, tears, and cursing, I wholeheartedly endorse this long overdue improvement. Having personally installed many racks on these new rails, I can tell you first-hand that it takes longer to unpack them than it does to install them. They're that good.



Figure 6-8. Closeup of a 7280R rail that attaches to the networking cabinet

High-Speed Ethernet

Arista made a name for themselves by making a big impact in the 10 Gigabit market. Since then, things have evolved a bit, and where 10 Gigabit used to be an exotic and expensive solution, it's now the standard, with higher speeds becoming ever more common. Here's a list of Ethernet speeds available on Arista switches as of early 2019:

NOTE

I should point out that in some cases there are multiple standards in play, and my descriptions might not apply to all of them, but from a generalized point of view, they should still be applicable.

10 Gigabit

The logical evolution of the 10/100/1000 bps standards, 10 Gbps Ethernet has become the new standard in data center speed. 10 Gbps is achieved over a single lane, meaning that it uses a single pair of fiber or solder traces.

25 Gigabit

25 Gigabit is an Arista-championed standard that was initially rejected by the IEEE for various political reasons that I won't get into here. Arista then got together with some of its biggest customers and partners such as Microsoft, Google, Broadcom, and Mellanox, who collectively became the 25 Gigabit Consortium. This consortium convinced the IEEE that 25 Gbps was a good idea, after which it was adopted as a standard, that being the first time that the IEEE ever accepted a standard proposed by an outside

body.

In a nutshell, 25 Gbps Ethernet allows for more than 2.5 times the speed of 10 Gbps over the same fiber as 10 Gbps because 25 Gbps uses a single lane, just like 10 Gbps does. This allows companies to upgrade significantly without having to replace their fiber plant.

25 Gbps Ethernet changed everything in ways that might not be immediately obvious, but as you continue down this list, we'll see why that's the case.

40 Gigabit

40 Gigabit Ethernet is essentially four 10 Gbps lanes with bits sent round-robin in hardware. It's technically a bit more involved than that, but what's important right now is that 40 Gbps is actually 4 times 10 Gbps. Note that this is not Etherchannel or LAG because the distribution is bit-round-robin, which results in a true 40 Gbps link.

50 Gigabit

Similar to the way that 40 Gbps Ethernet uses four 10 Gbps lanes, 50 Gbps Ethernet uses two 25 Gbps lanes.

There is also a design using 50 Gbps lanes, which gives the possibility of 100 Gbps (2 lanes), 200 Gbps (4 lanes), and so on.

100 Gigabit

There are multiple 100 Gbps standards, which I'll break into two generic groups that I'll call *old* and *new*.

Just as 40 Gbps is derived by bonding four 10 Gbps lanes, using the old standard, 100 Gbps is a result of bonding ten 10 Gbps lanes. It's actually common to use 12 lanes so that the 100 Gbps interface can be split into three 40 Gbps interfaces or 12 10 Gbps interfaces, but the IEEE standard does not include a 120 Gbps interface, so when configured for 100 Gbps, two of the lanes are essentially unused.

In the new paradigm, 100 Gbps is achieved by bonding four 25 Gbps lanes. Why is this so great? Because now customers can upgrade from their existing 40 Gbps networks to 100 Gbps without the huge cost associated with installing a completely new fiber plant. Not only that, but the 12 lanes of 10 Gbps method of deriving 100 Gbps speeds is expensive, produces a lot of heat, and is less efficient than the 4x25 Gbps method. Oh, and the 4x25 Gbps method can be implemented in a QSFP form factor, which means that Arista can make QSFP form-factor interfaces that support 10/25/40/50/100 Gbps—all in one interface! How cool is that? On the GAD-thinks-it's-cool-o-meter, it's about a 9 out of 10.

400 Gigabit

Originally designed with 16x25 Gbps lanes (yikes!), it is now looking like 8x50 Gbps will be the more likely solution, though Andy has said that he predicts 4x100 Gbps lanes will be the “only thing that actually makes sense.”

For 100 Gbps to take off there needs to be new optical standards, one of which is specified by the 100G Lambda Multi-Source Agreement, information about which you can view on YouTube.

By the way, OSFP stands for **O**ctal **S**mall **F**ormfactor **P**luggable, which is eight lanes of 56 or 112 Gbps (overhead being the reason they are not 50/100). I point this out because while the initialism makes perfect sense especially when compared to QSFP, the fact that OSFP is so close to the networking protocol OSPF makes me trip over the term every time I say it.

800 Gigabit

Using the OSFP with 100 Gbps lanes, 800 Gbps is possible optically, but as of 2018, the commitment is “not quite there yet” in silicon. You can view more information in this excellent YouTube video of Andy speaking on the matter.

Nonblocking Architecture

All Arista switches have nonblocking architectures (see [Chapter 5](#)), which means that Arista does not oversubscribe its switches. Have you ever bought a NetGear 8x1 Gbps switch only to discover that it could move only 2 Gbps of traffic? That's oversubscription (see [Chapters 3, 6, and 23](#)), and it's forever been a problem in other vendors' switches. Even the Arista 7010T is nonblocking, and it was pretty much designed to be a management switch.

FlexRoute

FlexRoute is the ability for certain Arista switches (generally those with “-R” suffixes like the 7280R) to support the full internet routing table with room for growth. This is something that was previously seen only in the realm of routers designed for the task. Arista changed the scope of what a networking switch was really capable of, much to the chagrin of the rest of the industry and its big iron (money) routers.

FlexRoute is covered in [Chapter 20](#).

AlgoMatch

Algomatch is a modern replacement for the decades-old ternary content-addressable memory (TCAM) architecture used in switches and routers. TCAMs are primarily known as a resource limitation in regards to Access Control Lists (ACLs) and Access Control Entries (ACEs), but they are also used in a variety of other aspects where masks are used for matching. AlgoMatch offers faster lookup time and increased capacity, which can be a big deal in environments with very large or numerous ACLs. AlgoMatch is a hardware feature and, as

such, is available only on certain model switches.

Optics

Arista optics are often less expensive than other vendors. In fact, I've seen Arista win bids based on the price of their optics alone. Even if the Arista switch was more expensive than the competitor's, the price of the Arista optics resulted in the bottom line being less than the competitor's quote. Hell, after five years at Arista I've seen other vendors lower their prices because Arista disrupted the market so much.

The price is nice, but a low price is useless if the item doesn't work. I'm happy to say that in my experience, they work as well as any other vendor's optics (if not better!).

GAD WHINES ABOUT OPTICS

Be warned that all the major vendors disallow the use of other vendor's optics. A Cisco SFP will not work in an Arista switch, nor will an HP SFP work in a Cisco switch. As a customer, I despised this game because it seemed like a way to force me into buying the same company's optics (vendor lock!) when another's might be less expensive, or even work better. Sadly, many of the big vendors do it, so I can't single out any one of them. I always found this practice to be a bit obnoxious because the SFP interface was designed for vendor interoperability (search for the terms *sfp multi source agreement* to read the original multivendor agreement on the SFP standard).

After working at Arista for many years, I have come to counter my own disdain for this practice because vendors don't do this only for vendor lock, but also to prevent the use of other vendors' often inferior products, which can (and often do) result in increased TAC cases. Given the huge influx of import knock-off optics in the past few years with flagrantly invalid vendor codes (even available on Amazon), I have seen for myself the kind of problems that can be caused by the use of out-of-spec optics, so my position has softened considerably on this issue.

Is there a way around this? Talk to your account team.

If you've done any data center networking in the past few years, you're

no doubt familiar with the SFP+, an example of which is shown in Figure 6-9.

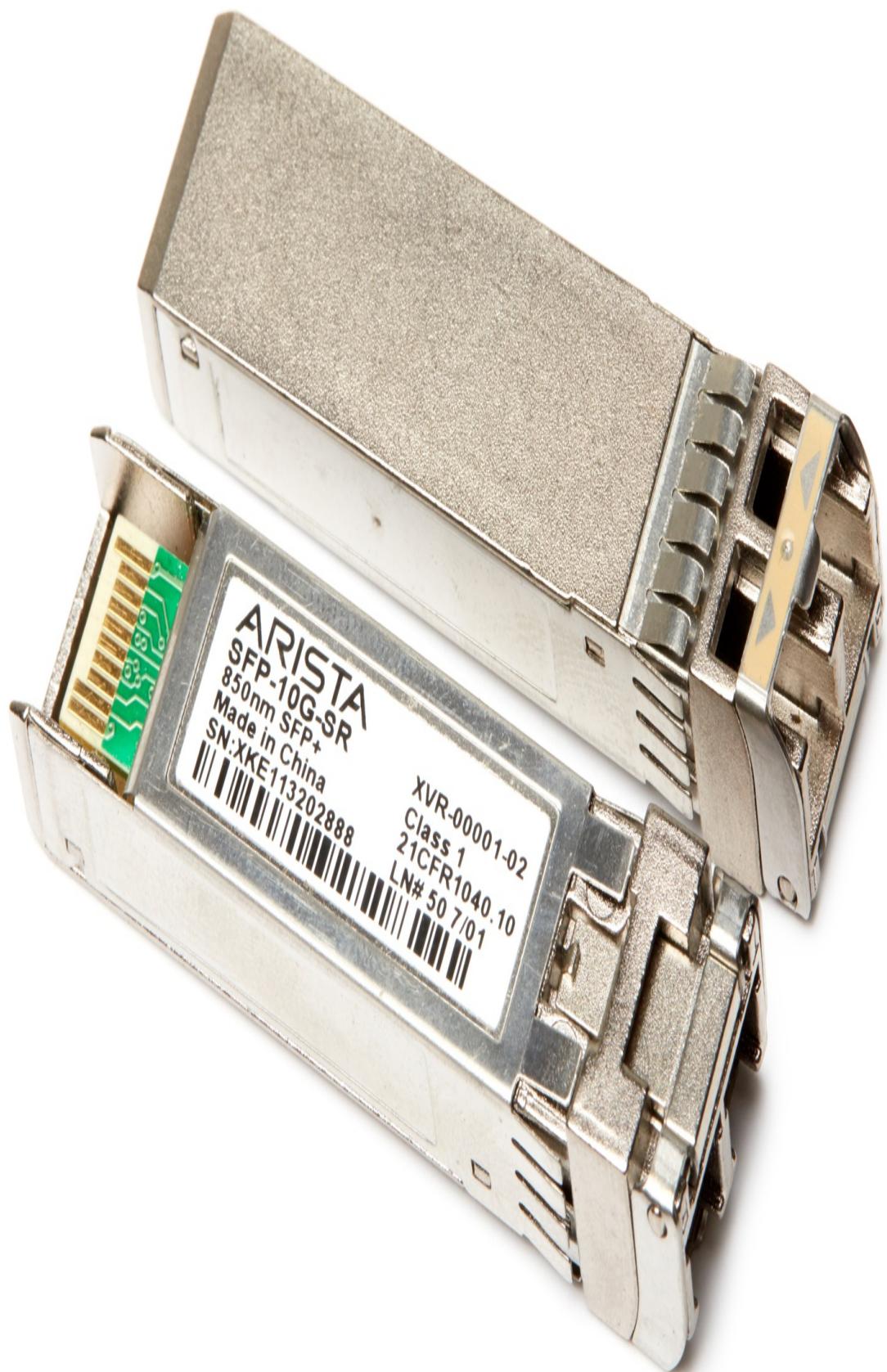


Figure 6-9. Arista SFP-10G-SR optics

The SFP+ is the evolution of the old Gigabit Interface Converter (GBIC) standard optic. The GBIC was limited to 1 Gbps, whereas the SFP+ supports 10 Gbps while being half the size. The XenPak (remember those?) supported 10 Gbps, but it was huge. With SFP+ optics, it is possible to fit 48 (or more) 10 Gbps interfaces within a single rack unit of space.

NOTE

Andy Bechtolsheim, founder of Arista, was one of the principal people responsible for the development and subsequent acceptance of the GBIC standard.

Also available are 40 Gbps QSFP+ modules, which are a slightly wider form factor than the SFP+. Both the QSFP+ and the SFP+ support twinax cables that allow for short-reach connectivity at a much lower cost than traditional optics. QSFP now supports 100 Gbps thanks to Andy championing 25 Gbps Ethernet, which, when run over four lanes, like 40 Gbps, produces 100 Gbps over the same cabling.

Software Products

Believe it or not, Arista Networks is primarily a software company. Yes, it made its name selling switches, but the majority of what Arista does is write great code. I once had a conversation with Andy (who is in charge of the hardware) about how great the hardware was and he waved his hand and said, “It’s the software that matters.” Of course, Ken Duda (who’s in charge of the software) said that the software was

nothing without the hardware. That's how maddeningly humble they both are.

EOS

One of the best features of all Arista switches (in my opinion) is the fact that they all run the same Extensible Operating System (EOS) code images. I did an install that had four different models of Arista switches. I needed to get them all on the same rev of code, so I downloaded the proper EOS version, put it on a USB thumb drive, walked to each switch, and copied the code onto flash. There is no need to figure out what hardware you have prior to downloading (though there are some minimum version and feature restrictions). There is no need for long, complicated filenames. There are no release trains, no versions for different feature sets, and generally just no pain when it comes to EOS. Every switch runs the same binary image. Simple is good, and Arista gets that. Spend some time upgrading Arista switches, and you'll detest upgrading on any other switch.

vEOS

vEOS is a version of EOS that runs as a virtual machine. There are actually two versions of vEOS now: *vEOS-Lab* and *vEOS-Router*. vEOS-Lab is what we used to call vEOS, meaning that it's free (guest registration required) and there is no support. vEOS-Router is designed to be purchased and supported and offers some cool features like Virtual Private Network (VPN) and routing, all with the availability of support contracts so that you can put it into production. vEOS-Router is the solution you're looking for if you're connecting into cloud architectures like Amazon Web Services (AWS) or Microsoft Azure,

whereas vEOS-Lab is what you're after if you want to build a lab on your laptop. I cover vEOS in [Chapter 33](#).

AnyCloud

AnyCloud is a solution that allows the customer to link to their cloud-based services (AWS, Azure, etc.) using the vEOS-Router and VPN. This means that virtual environments can now be securely linked into the customer's primary network using a network operating system that they already support: EOS.

cEOS

cEOS is the same idea as vEOS, only instead of running as a virtual machine, it runs in a *container*. What's a container? That's a pretty deep topic in and of itself, but for now consider a container to be like an ultra-lightweight virtual machine (VM) that starts almost instantly and has significantly less overhead than a traditional VM. For more information check out [Docker on EOS](#). I cover cEOS in [Chapter 32](#).

CloudVision

In the first edition of *Arista Warrior*, CloudVision was the name of what we generically call Extensible Messaging and Presence Protocol (XMPP) now. The XMPP feature remains in a greatly improved format, but the name CloudVision was given to a suite of software that does some pretty cool stuff. When first introduced, there was a lot of talk about it being a network management tool, though for this former NOC support guy, that term didn't make much sense to me.

CloudVision is mainly composed of two major components:

CloudVision Portal (CVP) and CloudVision eXchange (CVX). I cover both in more detail in [Chapter 15](#), but for now here's a brief introduction of the two main components:

CVP

CVP is generally what people are talking about when they use the term *CloudVision* because it's the software that involves all the slick graphical user interface (GUI) stuff you see in the marketing material. At its core, CVP is a configuration management tool, but it's capable of much more than simple configuration management. CVP is software that needs to be installed on a server (or cluster of servers).

CVX

CVX is really a version of vEOS running on a server with lots of memory that Arista switches connect to in order to share information. If you've seen drawings with terms like NetDB in which there's a master SysDB that other SysDBs connect to, that's accomplished with CVX.

Conclusion

As Arista continues to grow, there will no doubt be more great products to come along, both in the hardware and in the software realms. As someone who's been in the industry for decades and who has now worked at Arista for six years, I can tell you that it's tough for me to keep up, and I get to hear all the cool product announcements! As I sit here writing the final edits for the book, I'm already frustrated that I can't include them all.

Chapter 7. SysDB

One of the most impressive features of Arista’s Extensible Operating System (EOS) that makes it stand out from the competition is SysDB. Simply put, SysDB is a *System Database* on the switch that holds all of the state, variables, and any other important information so that processes can access it. Doesn’t sound too earth-shattering, now does it? Read on.

Traditionally, switches (and every other networking device out there) were built using *monolithic code*. Naturally, when I read the word *monolithic*, I think of apes dancing around the monolith in Stanley Kubrick’s masterpiece, *2001: A Space Odyssey*. That’s actually not a bad analogy, aside from the whole “spark of humanity” thing.

Networking devices have been around for decades now, and many of them are very mature products, running very mature code. Executives like to use the word *mature* to describe something that’s been around long enough to have all of the bugs worked out. Developers don’t always agree with the usage of this word.

The problem is that some of this code has been around for decades, too. In keeping with our monolithic analogy, imagine a switch that was first brought to market in, say, the year 2001. Now imagine that this switch is still in production 12 years later, and the software is up to around version 13, only instead of calling it version 13, let’s call it something

else, say, version 15. You know, because the number 13 is bad luck in many cultures, especially those that worship monolithic code.

Anyway, imagine that the initial software written for the switch consumed about 10 MB of disk space. Now imagine that every year, for the next 12 years, more and more features were added to that code. Imagine that the switch became so full of features that after those 12 years, the code grew to consume 100 MB of disk space.

Technology advances at a frightening pace and features expected as standard offerings today might not have been conceivable 12 years before. New hardware becomes more complex, which requires more complex code, all of which is added on to the original code base. If the original coders didn't anticipate 12 years of growth, they might not have made the original code easily expandable. Maybe, in the past 12 years, memory architectures changed. If you're as old as I am, you might remember a time when 640 K was more memory than a computer would ever need. Today, my Mac Pro currently has 128 GB of RAM. Things change. I can't imagine running a DOS machine today. As of this writing, Windows XP was all the rage 12 years ago. Would you want a switch based on Windows XP?

That's not all, though. Even if the code was written well and all those years of added routines were also written well, the fact remains that it's a single giant chunk of code that gets shoveled into memory. If you load an image that supports Spanning Tree Protocol (STP) but you don't need STP, the STP process is still in memory, sitting there, consuming space. Not only would that process consume space, but what would happen if that one piece of code crashed? With that code

being only a routine in the giant pile of monolithic code, the entire switch would crash. Bummer.

So, there must be a better way, right? Sure there is! Linux changed everything in the server world, and it's doing so in the networking world, as well. With Linux, the switch runs a kernel and a process manager. The process manager manages each process (hence the clever name) and restarts them if they crash. With this model, should STP crash, it wouldn't take the rest of the switch down with it. Or would it?

To properly protect a Linux system from misbehaving processes, each process should be written to use its own *user virtual address space*. Without getting into a server architecture discussion, understand that this type of memory is protected space that is addressable only by the process that owns it. If written using this model, should STP crash, only STP is affected; the rest of the switch continues to run unaffected.

There are two problems with this paradigm. First, many vendors don't properly utilize user virtual address space, so one misbehaving process often affects other processes. Second, if STP crashes, your network will reconverge when STP comes back online. This is because the STP lost all of its known network topology, timer values, and such when it crashed, so it must start as if the switch were just booted. Arista has solved both problems.

Arista code is well written and follows strict rules regarding user virtual address space. If the STP process crashes, no other process will be affected. More impressively, though, all state information for all processes is stored in the central database called *SysDB*. To further

explain, consider this: on an Arista switch, STP runs and stores all of its state information in SysDB. When any process starts, the first thing it does is go to SysDB to retrieve state information. If it finds a state, it uses it. If it does not find state information, it initializes. Let's take that to the next logical conclusion.

When a switch is booted, STP loads, checks SysDB (just booted, no state found), determines its state (through convergence, etc.), and then writes that state to SysDB. For some reason, STP crashes. Because STP is isolated, no other processes are affected. When STP crashes, the process manager restarts STP. STP loads, after which it immediately checks SysDB, where it finds state information. STP loads that information, and no convergence is necessary. This is a very big deal!

Don't believe me? Let's have a look.

WARNING

What I'm about to do is something that you shouldn't do in a production environment. Though the impact will be negligible (that's the point of this example), you should never take the complexities of a production environment for granted.

I've built a simple network between two switches, as shown in [Figure 7-1](#). There are two switches, Arista-1 and Arista-2. Arista-1 is the root, and Arista-2 is in its default configuration.

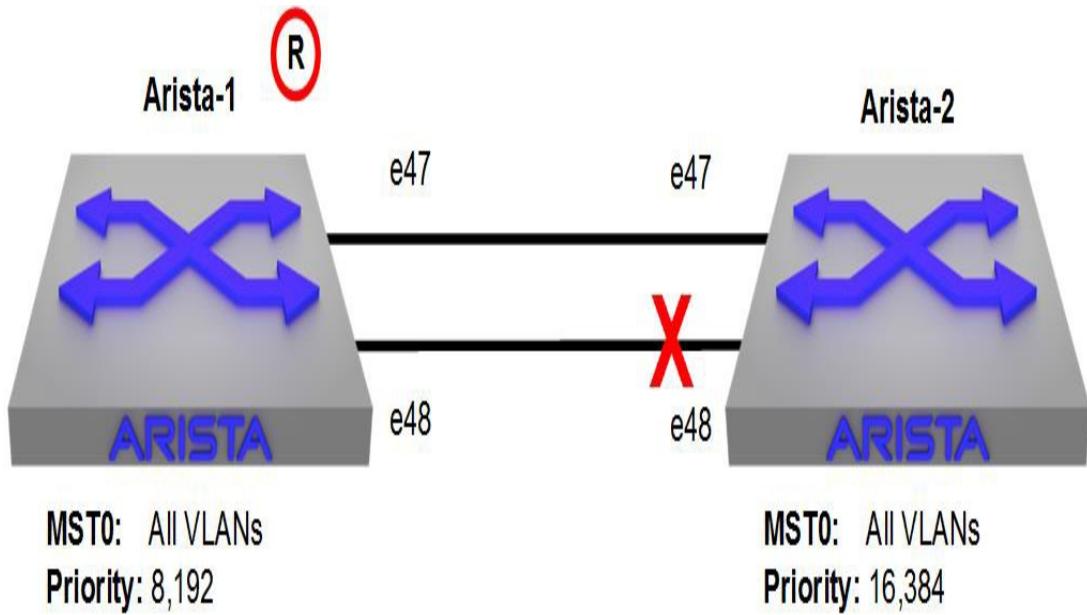


Figure 7-1. Simple network built for abusing EOS processes

With this network humming along nicely and STP performing as it should, let's go in and attempt to destroy it by killing STP on the Arista-1 switch. But first, let's look at what Spanning Tree sees on both sides. Here's Arista-1:

```

Arista-1(config)#sho spanning-tree
MST0
  Spanning tree enabled protocol mstp
  Root ID    Priority    8192
              Address    2899.3a06.696f
              This bridge is the root

  Bridge ID  Priority    8192  (priority 8192 sys-id-ext 0)
              Address    2899.3a06.696f
              Hello Time 2.000 sec  Max Age 20 sec  Forward Delay 15 sec

  Interface      Role      State      Cost      Prio.Nbr Type
  -----          -----      -----      -----      -----      -----
  Et47          designated  forwarding 2000      128.47  P2p
  Et48          designated  forwarding 2000      128.48  P2p

```

And here's the output from Arista-2:

```
Arista-2#sho spanning-tree
MST0
  Spanning tree enabled protocol mstp
  Root ID    Priority    8192
              Address     2899.3a06.696f
              Cost        0 (Ext) 2000 (Int)
              Port        47 (Ethernet47)
              Hello Time  2.000 sec  Max Age 20 sec  Forward Delay 15 sec

  Bridge ID  Priority    16384 (priority 16384 sys-id-ext 0)
              Address     2899.3a06.6b2b
              Hello Time  2.000 sec  Max Age 20 sec  Forward Delay 15 sec

  Interface   Role        State      Cost      Prio.Nbr  Type
  -----       -----      -----      -----      -----      -----
  Et47        root       forwarding 2000      128.47    P2p
  Et48        alternate  discarding 2000      128.48    P2p
```

As we can see, Arista-1 is the root, and E48 on Arista-2 is discarding, just as we expected.

Before we continue, I'd like to show you what the log entries look like when the Multiple Spanning Tree (MST) agent is initialized. To do that, I'm going to shut off MST entirely, restart it, and then show the logs:

```
Arista-1(config)#send log message [ Shutting down STP ]
Arista-1(config)#spanning-tree mode none

Jan 23 18:30:37 Arista-1 ConfigAgent: %SYS-6-LOGMSG_INFO: Message from
admin on
vty3 (10.0.0.100): [ Shutting down STP ]
Jan 23 18:30:45 Arista-1 StpTopology: %SPANTREE-6-DISABLED: STP is
disabled.
Jan 23 18:30:45 Arista-1 Stp: %SPANTREE-6-STABLE_CHANGE: Stp is now not
stable
Jan 23 18:30:45 Arista-1 Stp: %SPANTREE-6-INTERFACE_DEL: Interface
Ethernet47 has
```

```

been removed from instance MST0
Jan 23 18:30:45 Arista-1 Stp: %SPANTREE-6-INTERFACE_DEL: Interface
Ethernet48 has
been removed from instance MST0
Jan 23 18:30:46 Arista-1 Ebra: %LINEPROTO-5-UPDOWN: Line protocol on
Interface
Vlan50, changed state to down
Jan 23 18:30:46 Arista-1 Ebra: %LINEPROTO-5-UPDOWN: Line protocol on
Interface
Vlan50, changed state to up
Jan 23 18:31:15 Arista-1 Stp: %SPANTREE-6-STABLE_CHANGE: Stp state is now
stable
Arista-1(config)#show spanning-tree
Spanning-tree has been disabled in the configuration.

```

Now, I start MST. Pay attention to the entries for Ethernet 47 and 48 in the log:

```

Arista-1(config)#send log message [ Starting STP ]
Arista-1(config)#spanning-tree mode mst
Arista-1(config)#show log last 5 min
Jan 23 18:33:25 Arista-1 ConfigAgent: %SYS-6-LOGMSG_INFO: Message from
admin on
vty3 (10.0.0.100): [ Starting STP ]
Jan 23 18:33:31 Arista-1 StpTopology: %SPANTREE-6-MODE_CHANGE: Spanning
tree mode
is now mstp.
Jan 23 18:33:32 Arista-1 Ebra: %LINEPROTO-5-UPDOWN: Line protocol on
Interface
Vlan50, changed state to down
Jan 23 18:33:32 Arista-1 Stp: %SPANTREE-6-STABLE_CHANGE: Stp is now not
stable
Jan 23 18:33:32 Arista-1 Stp: %SPANTREE-6-ROOTCHANGE: Root changed for
instance
MST0: new root interface is (none), new root bridge mac address is
28:99:3a:06:69:6f (this switch)
Jan 23 18:33:32 Arista-1 Stp: %SPANTREE-6-INTERFACE_ADD: Interface
Ethernet48 has
been added to instance MST0
Jan 23 18:33:32 Arista-1 Stp: %SPANTREE-6-INTERFACE_ADD: Interface
Ethernet47 has
been added to instance MST0
Jan 23 18:33:32 Arista-1 Ebra: %LINEPROTO-5-UPDOWN: Line protocol on
Interface
Vlan50, changed state to up

```

```
Jan 23 18:33:32 Arista-1 Stp: %SPANTREE-6-INTERFACE_STATE: Interface
Ethernet48
  instance MST0 moving from discarding to learning
Jan 23 18:33:32 Arista-1 Stp: %SPANTREE-6-INTERFACE_STATE: Interface
Ethernet47
  instance MST0 moving from discarding to learning
Jan 23 18:33:33 Arista-1 Stp: %SPANTREE-6-INTERFACE_STATE: Interface
Ethernet48
  instance MST0 moving from learning to forwarding
Jan 23 18:33:33 Arista-1 Stp: %SPANTREE-6-INTERFACE_STATE: Interface
Ethernet47
  instance MST0 moving from learning to forwarding
```

The key entries here are in bold and show that when MST is initialized, it puts the interfaces into *learning* mode and then from *learning* into *forwarding*.

Now, let's wreak some havoc (as if that weren't enough havoc). There are a number of ways to kill a process in an Arista switch, but I'm going to use one of the most ungraceful ways possible and that's to kill everything that even resembles a process with the name "Stp" (MST is a form of the Spanning Tree Protocol and is thus controlled by the agent named Stp), and I'm going to kill it with extreme prejudice (that's what the "-9" means). Seriously, don't ever do this unless you're in a lab environment.

WARNING

That's worth another warning. Don't kill processes in a production switch unless you really know what you're doing, Arista Support tells you to, or you're looking to get fired anyway.

```
Arista-1(config)#send log message [ Killing STP ]
Arista-1(config)#bash sudo killall -9 Stp
Arista-1(config)#show log last 5 min
```

```
Jan 23 18:38:12 Arista-1 ConfigAgent: %SYS-6-LOGMSG_INFO: Message from
admin on
  vty3 (10.0.0.100): [ Killing STP ]
Jan 23 18:38:18 Arista-1 Stp: %AGENT-6-INITIALIZED: Agent Stp
  initialized; pid=8338
Jan 23 18:38:18 Arista-1 Stp: %SPANTREE-6-STABLE_CHANGE: Stp state is now
  stable
Jan 23 18:38:18 Arista-1 Stp: %SPANTREE-6-INTERFACE_ADD: Interface
  Ethernet47 has
    been added to instance MST0
Jan 23 18:38:18 Arista-1 Stp: %SPANTREE-6-INTERFACE_ADD: Interface
  Ethernet48 has
    been added to instance MST0
```

Notice what's missing? There are no *moving from discarding to learning* or *moving from learning to forwarding* messages. That's because the state was already known, so the interfaces didn't need to go through that process.

Within a second (much less, actually) of killing the Stp agent, the process manager restarted it, and thanks to SysDB, Spanning Tree (MST) never missed a beat. When the Stp agent started, it immediately requested the current state of the network from SysDB. Because there was a state found there, it simply used that instead of needing to start from an unknown network topology. Because this all happened well within the window of MST timers, the network didn't reconverge, and the other switch didn't even realize that there was a problem.

As a quick aside, if you begin playing around with agent killing, you'll likely discover the `agent` command. With this command, you can kill any agent without resorting to Bash commands. To kill the Stp process, the following command (which is undocumented in newer versions of code like 4.17.2F) would have the same effect as the Bash `sudo killall -9 Stp` command we used earlier:

```
VM-SW3#agent stp terminate
Stp was terminated
```

That's an undocumented command on EOS 4.21.1F, and it's likely undocumented for a reason, so you probably shouldn't do that. Other options include `environment` and `shutdown`, both of which you should also leave alone unless Arista TAC or your sales/customer engineer recommends otherwise.

We've seen what happens when we kill Spanning Tree, which illustrates the power and benefit of SysDB, but what happens if SysDB itself dies a horrible, unnatural death? Let's find out! But first, let's save our configuration because if years of video games have taught me anything, it's to save your game before a big boss battle:

```
Arista-1(config)#wri
Copy completed successfully.
```

OK, let's run into that cave with a resounding scream of *Leeroy Jenkins!*

```
Arista-1(config)#bash sudo killall -9 Sysdb
Arista-1(config)#
Connection to Arista-1 closed.
gad@[Lab]:~$
```

Uh...that can't be good. A couple of seconds later, though, I can get right back in. Here's what I would have seen if I'd done that on the console:

```
Arista-1#bash sudo killall -9 Sysdb
Arista-1#Jan 23 18:51:29 Arista-1 Lldp: %FWK-3-SOCKET_CLOSE_REMOTE:
Connection to
Sysdb (pid:9129) at tbl://sysdb/+n closed by peer (EOF)
Jan 23 18:51:29 Arista-1 Bfd: %FWK-3-SOCKET_CLOSE_REMOTE: Connection to
```

```

Sysdb
  (pid:9129) at tbl://sysdb/+n closed by peer (EOF)
Jan 23 18:51:29 Arista-1 Capi: %FWK-3-SOCKET_CLOSE_REMOTE: Connection to
Sysdb
  (pid:9129) at tbl://sysdb/+n closed by peer (EOF)
Jan 23 18:51:29 Arista-1 Bfd: %FWK-3-MOUNT_PEER_CLOSED: Peer closed
socket
  connection. (tbl://sysdb/+n-in)(Sysdb (pid:9129))
Jan 23 18:51:29 Arista-1 SuperServer: %FWK-3-SOCKET_CLOSE_REMOTE:
Connection to
  Sysdb (pid:9129) at tbl://sysdb/+n closed by peer (EOF)
Jan 23 18:51:29 Arista-1 Bfd: %FWK-3-MOUNT_CLOSED_EXIT: Process exiting.

[-- much output removed --]

Jan 23 18:51:37 Arista-1 Sysdb: %AGENT-6-INITIALIZED: Agent 'Sysdb' initialized; pid=11321
Jan 23 18:51:40 Arista-1 ConfigAgent: %AGENT-6-INITIALIZED: Agent 'ConfigAgent'
initialized; pid=11336
Jan 23 18:52:06 Arista-1 ConfigAgent: %SYS-5-CONFIG_E: Enter
configuration mode
  from console by root on UnknownTty (UnknownIpAddress)
Jan 23 18:52:11 Arista-1 ConfigAgent: %SYS-5-CONFIG_I: Configured from
console
  by root on UnknownTty (UnknownIpAddress)
Jan 23 18:52:18 Arista-1 StageMgr: %AGENT-6-INITIALIZED: Agent 'StageMgr'
initialized; pid=11335
Jan 23 18:52:18 Arista-1 Fru: %AGENT-6-INITIALIZED: Agent 'Fru'
initialized;
  pid=11338
Jan 23 18:52:18 Arista-1 Launcher: %AGENT-6-INITIALIZED: Agent 'Launcher'
initialized; pid=11339

```

In a nutshell, after killing the process at the heart of the entire switch, it recovered gracefully. To be fair, *gracefully* means that all the processes needed to reinitialize, and the end result was the same as rebooting the switch, but consider this: on my Arista 7280R, it takes 4 minutes and 15 seconds from the point at which I enter the `reload now` command to the point when I get a login. If I kill SysDB, I get a login prompt again after about 30 seconds.

Let's have some fun and use that to our advantage. Because the *running-config* resides in SysDB, and SysDB is reinitialized when it's killed, I'm going to load a completely new configuration on the switch without rebooting. Ever been frustrated by needing to reboot in order to load a completely new configuration? This isn't a total cure, but it sure beats rebooting.

First, I save the current *startup-config* to a file named *OLD-Config*:

```
Arista-1#copy startup-config OLD-Config
Copy completed successfully.
```

Next, I remove the current *startup-config* with the `write erase` command:

```
Arista-1#write erase
Proceed with erasing startup configuration? [confirm]<cr>
Arista-1#
```

I've written a simple configuration file that has nothing except the defaults and a changed hostname, just to show that the new configuration actually is loaded. The new hostname will be *GAD-1*. The filename for this configuration is *GAD-Config*:

```
Arista-1#copy flash:GAD-Config startup-config
Copy completed successfully.
```

At this point, my switch is running according to the configuration loaded in the *running-config*, but when it boots, it will load the code in the *startup-config*, which has my new hostname installed. Remember, if I were to reboot the switch, it would take almost two minutes. Instead, I'll kill SysDB from the console and watch what happens:

```

Arista-1#bash sudo killall -9 Sysdb
Arista-1#Jan 23 19:07:49 Arista-1 StageMgr: %FWK-3-SOCKET_CLOSE_REMOTE:
  Connection to Sysdb (pid:4137) at tbl://sysdb/+n closed by peer (EOF)
Jan 23 19:07:49 Arista-1 Fru: %FWK-3-SOCKET_CLOSE_REMOTE: Connection to
  Sysdb (pid:4137) at tbl://sysdb/+n closed by peer (EOF)
Jan 23 19:07:49 Arista-1 ConfigAgent: %FWK-3-SOCKET_CLOSE_REMOTE:
  Connection
    to Sysdb (pid:4137) at tbl://sysdb/+n closed by peer (EOF)
Jan 23 19:07:49 Arista-1 Launcher: %FWK-3-SOCKET_CLOSE_REMOTE: Connection
  to Sysdb (pid:4137) at

[--- Processes whining about SysDB removed ---]

Jan 23 19:20:27 Arista-1 SandMact: %FWK-3-MOUNT_PEER_CLOSED: Peer closed
socket
  connection. (tbl://sysdb/+n-in)(Sysdb (pid:9618))
Jan 23 19:20:27 Arista-1 SandMact: %FWK-3-MOUNT_CLOSED_EXIT: Process
  exiting.
Jan 23 19:20:27 Arista-1 SandFap: %FWK-3-SOCKET_CLOSE_REMOTE: Connection
  to Sysdb
  (pid:9618) at tbl://sysdb/+n closed by peer (EPIPE when writing)

GAD-1 login:
Jan 23 19:23:56 GAD-1 SandL3Unicast: %HARDWARE-3-DISRUPTIVE_RESTART:
  Non-disruptive restart of SandL3Unicast forwarding agent failed. All
  hardware
  entries in Routing table will be deleted and re-programmed in the
  hardware.
  Reason: agent was restarted twice in quick succession.

GAD-1 login:

```

And just like that, in 15 seconds flat, I've reloaded my Arista switch with a new configuration. Try that with a monolithic code-based switch from another vendor! To be completely fair, the time until all interfaces are up, all protocols settle, and things like Multi-Chassis Link Aggregation Group (MLAG) stabilize will depend on the switch platform and the configuration.

NOTE

If you decide to play around with killing SysDB, be warned that, as stated, it has the same

effect as rebooting the switch. This means that unsaved changes to the *running-config* will be lost. Because killing SysDB isn't *really* graceful, there's no warning about unsaved changes. But if you're doing things like killing SysDB, you must know what you're doing, right?

Oh—did you notice that scary message at the end? Here it is again:

```
Jan 23 19:23:56 GAD-1 SandL3Unicast: %HARDWARE-3-DISRUPTIVE_RESTART:  
  Non-disruptive restart of SandL3Unicast forwarding agent failed. All  
  hardware  
  entries in Routing table will be deleted and re-programmed in the  
  hardware.  
  Reason: agent was restarted twice in quick succession.
```

Yikes! Be warned: when you play around with killing agents, bad things can (and probably will) happen, so don't do that!

Oh, and I should probably mention that on EOS releases 4.14 and higher, there is a very sexy command called `config replace`. This allows you to specify a new *running-config* and load it without reloading the switch, without killing SysDB, and without interrupting processes that have not changed their configurations! Here I am loading my original configuration back on my 7280R running version 4.21.1F:

```
GAD-1#configure replace flash:OLD-Config  
Arista-1#
```

Note that the only obvious change is the hostname, but it loaded a new configuration in seven seconds with no downtime. That's much better than killing SysDB! For more details on the `configure replace` command, see [Chapter 9](#).

SysDB is the heart of Arista's EOS, and as I've hopefully shown in this chapter, the resiliency of SysDB along with the power of Linux makes for an amazingly robust switch operating system that almost can't be killed.

As a last note, SysDB is an agent, so it resides in memory. Its contents are built after every boot, which means that it does not survive a reboot (that's what the *startup-config* is for). Finally, SysDB is actually a very robust agent and the last time I asked had never spontaneously crashed in a customer network, so you don't need to worry about seeing that in the real world. Just keep me away from your switches, and you'll be fine.

Conclusion

In [Chapter 5](#), I said that the fabric was the heart of the switch, and that's true when it comes to the hardware. When it comes to the software, though, the heart of EOS has to be the concept of SysDB, the concept upon which almost everything else in the operating system is based.

Chapter 8. Introduction to EOS

The operating system for Arista switches is called the Extensible Operating System, or EOS for short. Arista describes EOS as “...the interface between the switch and the software that controls the switch and manages the network.” This is sort of like Apple’s macOS in that what you see is actually a kind of Unix (actually a derivative of FreeBSD in OS X) shell; Unix is doing all the heavy lifting behind the scenes. Arista switches run Unix (actually Fedora Core Linux) natively, but to make them easier for nonprogrammers to understand, EOS makes them look more like industry-standard networking devices.

The word *extensible* means “capable of being extended,” and EOS was designed from the ground up to allow third-party development of add-ons. This is a first in the networking world and is a big departure from traditional proprietary operating systems.

Arista is a big believer in open standards, and there are no proprietary protocols found in EOS that I am aware of. Even features such as Multi-Chassis Link Aggregation Group (MLAG) and Virtual Address Resolution Protocol (VARP), both Arista developments, use behaviors found in existing open-standard protocols, the details of which we cover later in [Chapter 18](#) and [Chapter 19](#).

Perhaps even more impressively, Arista allows the user to access the

underlying Linux operating system (OS) and even to write Python and Bash scripts that can control the switch. This is a significant difference from other vendors that advertise that their switches run a derivative of Linux. Although those switches might be based on Linux, you might not be able to get to it, and all the power of Linux remains just out of your grasp while you struggle with a new OS that's almost like the one you've grown accustomed to.

On an Arista switch, the switch is running Linux. In fact, the switch is actually a Linux server with custom hardware that has a lot of interfaces. I'll show you just how true that is in this book.

Some customers have even built their own interface to the switches, bypassing EOS altogether. Arista offers APIs for programmers toward this end and even hosts a website, [EOS Central](#), where customers can share their ideas or scripts with other users (registration is required).

SysDB

Arista switches have, at their heart, a database called SysDB (Chapter 7). This database contains the state information and settings for the switch, organized in such a way that every Arista process or *agent* can access it with ease.

NOTE

SysDB is a database in the computer science sense of the term, meaning that it is a collection of data in a single place. It is not a database like MySQL with a query language that you can use to access and update tables; rather it's more like a hierarchical tree of state information that can be accessed and updated by Arista agents.

Traditional networking hardware uses a monolithic software architecture. This means that there is a pretty significant risk of a bug in one section of code affecting or even bringing down the entire device. Furthermore, updating one section of code can be difficult because there can be repercussions in other areas unforeseen by the developer.

Arista's EOS is more modular. Not only does EOS separate the networking state from the processing, but drivers, processes, management, and even security patches run in user address space, not in memory shared with the kernel. This means that almost any process can be restarted without affecting the state of the network that it controls. This also means that any agent can be upgraded without affecting traffic flow. Agents can be added with ease, and faults are isolated to their individual user spaces. Should one process crash, it cannot affect the rest of the system. Should the process crash or lock up, the EOS process manager (ProcMgr) can restart it without affecting other modules and without affecting the networking state. Because SysDB is just a hierarchy of state information and contains no application code, it is extremely reliable.

Because EOS is so modular, the drivers for the Application-Specific Integrated Circuits (ASICs) are merely agents. Because of this, EOS is the same for every switch Arista makes. There are no release trains, no hardware-specific downloads, and no hours wasted trying to find the right code. If you want EOS 4.21.1F, you download the code named *EOS-4.21.1F.swi*. If you have a 384-port 7508R chassis switch or a 72-

port 7280R fixed-configuration switch, the software image is the same.

THE SINGLE IMAGE EXPLAINED

Have you ever had the pleasure of trying to get all of your networking equipment running on the same version of code only to discover that you can't stemming from myriad reasons usually having to do with different equipment from the same vendor actually coming from different business units (BUs)? Arista has one BU, and it's called Arista. Because of this and because of the way that EOS is constructed, you can put the same rev of EOS on every Arista switch in your data center. Well, mostly.

The detail that makes this mostly true is that any current rev of EOS will run on any current piece of Arista hardware. That's still a very big deal, so let me explain why this is a limitation.

EOS works on any current switch because it loads a driver at boot time that interfaces with the ASIC in the switch. If EOS discovers a Jericho ASIC, it loads the appropriate driver and then continues on. Because some ASICs are no longer current and have been End-of-Lived (EoL), their support has been removed from EOS in order to save space. That's why you can't run, for example, EOS-4.21.1F.swi on an old Arista 7148 switch.

Similarly, when Arista stopped updating EOS version 4.12, the Jericho ASIC didn't yet exist, so that old rev of code will not work on a newer model switch like the 7280R. Any current revision of EOS will work on any current Arista switch, though.

Using EOS

If you've used an industry-standard switch, you can use an Arista switch. As soon as you log in to an Arista switch, you'll recognize the look and feel because it's very similar to, for example, IOS. The important distinction is that it is not IOS because the internals have been completely written from scratch. Only the command-line interface (CLI) is similar.

One of the things that frustrated me when I hooked up my first Arista switch was the fact that telnet is not enabled. In its default configuration, Secure Shell (SSH) is the only means allowed to remotely access EOS. Certainly, console access is allowed, and telnet

can be enabled, but in this time of Payment Card Industry (PCI) regulations, Sarbanes-Oxley, and countless other security-centric requirements, keeping telnet disabled is a good idea. As I joke in my training classes, “it’s 20-freaking-19; stop using telnet!”

For my examples, I’m connecting through the console (through a console server). Logging into EOS is as simple as it is with IOS:

```
Arista-7280 login: admin
Last login: Fri Sep 16 15:33:59 on ttys0
Arista-7280>
```

With only simple login authentication configured, I am dropped into *EXEC* mode, which should look pretty familiar. At this point, I can access *Privileged EXEC* mode by using the `enable` command:

```
Arista-7280>en
Arista-7280#
```

The prompt has changed, as I’d expect it to, and I now have the power. At this point, I’ll add my own username because I like to be accountable for my actions. Actually, that’s a lie. I’m so lazy that I like to have the same username and password on every device I’ve ever configured so that I don’t need to remember them. OK, so that’s not true either, but I thought I’d configure a username just to show how the process of configuration works in EOS.

Again, this is just like the industry standard. Here, I configure from the terminal using the `config terminal` command, then configure my username, and then exit:

```
Arista-7280#conf
```

```
Arista-7280(config)#
```

At this point, I am in *global configuration* mode. By the way, notice how I didn't type **conf t**? Although the CLI will take **configure terminal**, the **terminal** part is no longer necessary. Why not? I often ask the students in my class what options there are other than *terminal*. Usually only the old guys know the answer because we haven't used any of them in years.

I should point out, though, that this has changed in the past couple of years with the advent of features such as **config replace** and **config session** (Chapter 9), which makes it even more interesting to me that you can still just type **conf** to get into configuration mode. Let's get back to our example:

```
Arista-7280(config)#username GAD secret ILikePie
Arista-7280(config)#exit
Arista-7280#exit
Arista-7280 login: GAD
Password: ILikePie
Arista-7280>en
Arista-7280#
```

EOS does not support clear-text passwords, and there is no command **service password-encryption** like there is in other operating systems. EOS no longer uses MD5 for encryption of passwords, either, opting instead for the much more powerful and secure SHA512. As such, the running configuration for my user now looks like this:

```
username GAD secret sha512
$6$dxUPdkGx22Cn5pUd$mok04fVNTa0Z3vvFCvrpNboushWBdiJGi
waBcuiLPCAclETvUt.rfENIRbRfFv9TAZmn2JP5u10BErznrughXu1
```

Now let's try entering an abbreviated command:

```
Arista-7280#ro
% Incomplete command
```

Because that's ambiguous, I get an Incomplete command error. I can, however, find out what commands are available in one of two ways. First, I can press the "?" (question mark) key. This will give me a list of available commands that match what I've entered so far:

```
Arista-7280(config)#ro?
role  route-map  router
```

I should also point out that EOS does not show you where you made an error if you make one. IOS will show you a little carrot and point out where you fat-fingered the command. Here's an example from an old Cisco 3750:

```
3750(config)#routter
^
% Invalid input detected at '^' marker.
```

EOS does not do this, instead just giving a generic Invalid Input message:

```
Arista(config)#routter
% Invalid input
```

Back to figuring out commands from the CLI, I can also press the Tab key, at which point the switch will respond with the longest match based on what I've typed so far:

```
Arista-7280(config)#rou<TAB>
Arista-7280(config)#route
```

Note that I did not type the word `route`; the switch inserted that via *autocomplete* when I pressed Tab. At this point I decided that it's the `router` command I was looking for, so I added the `r` and then pressed “?”. The switch then recognized that `router` is a command and listed the possible associated keywords:

```
Arista-7280(config)#router ?
  bgp                  Border Gateway Protocol
  general              Protocol independent routing configuration
  igmp                 Internet Group Management Protocol (IGMP)
  isis                 Intermediate System - Intermediate System (IS-IS)
  kernel                Routes installed by kernel
  msdp                 Multicast Source Discovery Protocol (MSDP)
  multicast             Multicast routing commands
  ospf                 Open Shortest Path First (OSPF)
  ospfv3                OSPF Version 3
  pim                  Protocol Independent Multicast
  rip                  Routing Information Protocol
  traffic-engineering  traffic-engineering global config
```

I chose one of these protocols, after which the switch put me into protocol-specific mode and altered the command line to show where I was:

```
Arista-7280(config)#router ospf 100
Arista-7280(config-router-ospf)#

```

As with the industry standard, typing `exit` (or its nonambiguous abbreviation) got me out of the current level and popped me back up one level:

```
Arista-7280(config-router-ospf)#ex
Arista-7280(config)#

```

NOTE

You don't need to type `exit`. If you want to work in another mode, you can just type in the

command, and EOS will switch modes for you, assuming that it can figure out the proper mode.

By typing `end`, or Control-Z, I was able to exit configuration mode entirely:

```
Arista-7280(config-router-ospf)#end
Arista-7280#
```

At this point I'd like to add that the EOS CLI recognizes some Emacs control characters. I am constantly amazed that even networking guys with decades of experience are unaware of these simple CLI key combinations. The following Control key combinations will have the effects listed:

Control-A

Moves the cursor to the beginning of line

Control-E

Moves the cursor to the end of line

Control-B

Moves the cursor back one character (same as left arrow)

Control-F

Moves the cursor forward one character (same as right arrow)

Esc-B

Moves the cursor back one word

Esc-F

Moves the cursor forward one word

Entering the interface command, followed by the interface name, puts

you into interface configuration mode. Interface configuration mode is similar to IOS, but more robust.

NOTE

Front-panel interfaces on Arista switches are all Ethernet and are not named in accordance with their speed or type. All Ethernet interfaces on a fixed configuration switch have the name `ethernet interface#` (the space is optional). Ethernet interfaces on modular switches have the name `ethernet slot#/interface#`. For example, the first interface on an Arista 7150 is `ethernet1`, or `e1` for short. The first interface in slot number three of an Arista 7508R modular switch is `ethernet 3/1` or `e3/1`. On interfaces with lanes such as a 40 Gbps interface, an additional field is added so that the interface name becomes `slot#/interface#/laneid`. Thus, on a 7508R with a 40 Gbps blade in slot3, the first interface on the blade would be `ethernet 3/1/1`.

Here, I've entered the interface configuration mode for `ethernet 1`:

```
Arista-7280#conf
Arista-7280(config)#int e1
Arista-7280(config-if-Et1)#
```

One of the nice enhancements to EOS is that it has no interface range command. To configure multiple interfaces at one time, simply enter them separated by either a hyphen (for a range) or a comma (for a list). In the example that follows, I've entered configuration mode for the interfaces `e1`, `e2`, `e3`, and `e10`. I've included the first three as a range:

```
Arista-7280(config)#int e1-3, e10
Arista-7280(config-if-Et1-3,10)#
```

Check out the command prompt. It doesn't show just "range," but rather shows what interfaces are being configured. I love this feature

because I can never remember what I typed a few short seconds ago.

Here's a super-cool feature that's been around for a while that was shown to me by Rich Parkin, who gets named credit because I'd been at Arista for five years and didn't know it was possible only to be shown by the new guy who blew me away with this amazing feature. Imagine that you need to apply IP addresses to a range of interfaces. Wouldn't it be cool if you could do that in some sort of cool automated fashion? Check this out:

```
Arista(config)#int e1-4
Arista(config-if-Et1-4)#no switchporth
Arista(config-if-Et1-4)#ip address 10.1.{1,4}.1/24
```

Whoa! Curly braces? In networking?

NOTE

I joke in my classes all the time that many networking people, upon seeing a curly brace, immediately check out because it looks like programming. If curly braces freak you out, make sure to read my rant about programming in [Chapter 30](#).

Let's see what this command has done by looking at the *running-config*:

```
Arista(config-if-Et1-4)#sho run int e1-4
interface Ethernet1
    no switchport
    ip address 10.1.1.1/24
interface Ethernet2
    no switchport
    ip address 10.1.2.1/24
interface Ethernet3
    no switchport
```

```
  ip address 10.1.3.1/24
interface Ethernet4
  no switchport
  ip address 10.1.4.1/24
```

How cool is that? The third octet has been applied starting at one and incrementing until it got to four. That's the meaning of `{1,4}`. But wait; there's more! The first two fields that we used are called *start* and *end*. You can also add a field to this bit of curly-braced madness, which is called a *step*. This will allow you to do things like put in only even numbers. Let's clear the IP addresses and put in the same IP addresses, but this time the third octet will have the numbers 2,4,6,8:

```
Arista(config-if-Et1-4)#ip address 10.1.{2,8,2}.1/24
Arista(config-if-Et1-4)#sho run int e1-4
interface Ethernet1
  no switchport
  ip address 10.1.2.1/24
interface Ethernet2
  no switchport
  ip address 10.1.4.1/24
interface Ethernet3
  no switchport
  ip address 10.1.6.1/24
interface Ethernet4
  no switchport
  ip address 10.1.8.1/24
```

Want odd numbers? Start with 1, go to 7, and step 2:

```
Arista(config-if-Et1-4)#ip address 10.1.{1,7,2}.1/24
Arista(config-if-Et1-4)#sho run int e1-4
interface Ethernet1
  no switchport
  ip address 10.1.1.1/24
interface Ethernet2
  no switchport
  ip address 10.1.3.1/24
interface Ethernet3
  no switchport
```

```
  ip address 10.1.5.1/24
interface Ethernet4
  no switchport
  ip address 10.1.7.1/24
```

But wait; there's more! You can even use this functionality to name your interfaces—and not only that, you can format the output with padding by adding a pad field. Here I'm adding a description to each interface that reads DC1-Row-XX where *XX* is a two-digit, right-justified, zero-padded field:

```
Arista(config-if-Et1-4)#desc DC1-Row-{1,7,2,2}
Arista(config-if-Et1-4)#sho run int e1-4
interface Ethernet1
  description DC1-Row-01
  no switchport
  ip address 10.1.1.1/24
interface Ethernet2
  description DC1-Row-03
  no switchport
  ip address 10.1.3.1/24
interface Ethernet3
  description DC1-Row-05
  no switchport
  ip address 10.1.5.1/24
interface Ethernet4
  description DC1-Row-07
  no switchport
  ip address 10.1.7.1/24
```

How cool is that? Pretty darn cool, I'd say.

Back to interface ranges, be warned that the prompt can get a bit unwieldy with long lists of interfaces because they will all show up in the command prompt:

```
Arista-7280(config)#int e1,3,5,7,9,11,13,15,17,19
Arista-7280R(config-if-Et1,3,5,7,9,11,13,15,17,19)#no ip address
Arista-7280R(config-if-Et1,3,5,7,9,11,13,15,17,19)#no description
```

```
Arista-7280R(config-if-Et1,3,5,7,9,11,13,15,17,19)#switchport
Arista-7280(config-if-Et1,3,5,7,9,11,13,15,17,19)#description Odd
```

Another nice feature of EOS is that I don't need to exit configuration mode to run exec mode commands. I just run them, and EOS figures it all out. There is no **do** command necessary with EOS:

```
Arista-7280R(config-if-Et1,3,5,7,9,11,13,15,17,19)#sho active
interface Ethernet1
    description Odd
interface Ethernet3
    description Odd
interface Ethernet5
    description Odd
interface Ethernet7
    description Odd
interface Ethernet9
    description Odd
interface Ethernet11
    description Odd
interface Ethernet13
    description Odd
interface Ethernet15
    description Odd
interface Ethernet17
    description Odd
interface Ethernet19
    description Odd
```

By the way, that `show active` command is the same as doing a `show running-config section-name` for the current section when in configuration mode, which in this case happens to be a range of interfaces.

Ooh, here's another cool Aristacism (I totally made up that word. See [Chapter 33](#)): when specifying interface ranges, you can use the dollar sign to indicate the last interface within the range. On an Arista 7010T, this results in the following:

```
Arista-7010T(config)#int e1-$  
Arista-7010T(config-if-Et1-52)#
```

On a switch that has interfaces with lanes or blades (or both) this will appear to not work:

```
Arista-7280R(config)#int e1-$  
% Incomplete command
```

This is because some of the interfaces on this switch have lanes and thus are referenced (for example) as `int e50/1`. To use the dollar sign in the range on a switch like this you must include all of the possible fields:

```
Arista-7280R(config)#int e1-$/$  
Arista-7280R(config-if-Et1-49/1,50/1,...,53/1,54/1)#
```

Notice the ellipsis in that range prompt? The developers have made it so that very long ranges are abbreviated to prevent unusable prompts.

Technically, any mode can run commands from any parent mode. If you enter commands from a different mode at the same level (interface- and protocol-specific mode, for example), the mode will switch accordingly. You cannot, however, execute child mode commands from a parent mode. For example, you cannot execute interface-specific commands from within the global configuration mode.

I was able to pipe the output of `show run` to `include`, which outputs only what I wanted to see. This behavior is similar to the `grep` command in Linux. There are a bunch of options for piping in EOS, which you can see by entering `| ?` (vertical bar, question mark) after

any **show** command:

```
Arista-7280(config)#sho run | ?  
LINE      Filter command by common Linux tools such as grep/awk/sed/wc  
append    Append redirected output to URL  
begin     Begin with the line that matches  
exclude   Exclude lines that match  
include   Include lines that match  
json      Produce JSON output for this command  
no-more   Disable pagination for this command  
nz        Include only non-zero counters  
redirect  Redirect output to URL  
section   Include sections that match  
tee      Copy output to URL
```

Though **include** is similar to **grep**, it differs in a significant way. This distinction is important when stacking pipes. The command **sho int | inc Ethernet** will output the following:

```
Arista-7280(config)#sho int | inc Ethernet  
  
Ethernet1 is up, line protocol is up (connected)  
  Hardware is Ethernet, address is 001c.7390.93d0 (bia 001c.7390.93d0)  
    Ethernet MTU 9214 bytes , BW 10000000 kbit  
Ethernet2 is down, line protocol is notpresent (notconnect)  
  Hardware is Ethernet, address is 001c.7390.93d1 (bia 001c.7390.93d1)  
    Ethernet MTU 9214 bytes , BW 10000000 kbit  
Ethernet3 is down, line protocol is notpresent (notconnect)  
  Hardware is Ethernet, address is 001c.7390.93d2 (bia 001c.7390.93d2)  
    Ethernet MTU 9214 bytes , BW 10000000 kbit  
[--- output truncated ---]
```

Now suppose that I wanted to further filter that output and include only the lines that contain the word *Hardware*. My inclination would be to add another pipe with another include, like this: **sho int | inc Ethernet | inc Hardware**. The problem is, this doesn't work properly (if at all):

```
Arista-7280#sho int | inc Ethernet | inc Hardware
```

```
Ethernet MTU 9214 bytes , BW 1000000 kbit
Ethernet MTU 9214 bytes , BW 10000000 kbit
```

If I change my `includes` to `greps`, it works as I desire:

```
Arista-7280#sho int | grep Ethernet | grep Hardware
Hardware is Ethernet, address is 001c.7390.93d0 (bia 001c.7390.93d0)
Hardware is Ethernet, address is 001c.7390.93d1 (bia 001c.7390.93d1)
Hardware is Ethernet, address is 001c.7390.93d2 (bia 001c.7390.93d2)
Hardware is Ethernet, address is 001c.7390.93d3 (bia 001c.7390.93d3)
Hardware is Ethernet, address is 001c.7390.93d4 (bia 001c.7390.93d4)
Hardware is Ethernet, address is 001c.7390.93d5 (bia 001c.7390.93d5)
Hardware is Ethernet, address is 001c.7390.93d6 (bia 001c.7390.93d6)
Hardware is Ethernet, address is 001c.7390.93d7 (bia 001c.7390.93d7)
```

So why does this happen? Whereas `include` is part of the command-line interpreter, `grep` is a Linux command. Piping is a Linux function, so EOS behaves more like Linux when stacking Linux commands the way I have.

Note that you can also pipe to some Linux commands even though they're not included in the help output. For example, `egrep` is not included in the output, but we can use it from the CLI:

```
Arista-7280(config)#sho int | egrep -i hardware
Hardware is Ethernet, address is 001c.7390.93d0 (bia 001c.7390.93d0)
Hardware is Ethernet, address is 001c.7390.93d1 (bia 001c.7390.93d1)
Hardware is Ethernet, address is 001c.7390.93d2 (bia 001c.7390.93d2)
Hardware is Ethernet, address is 001c.7390.93d3 (bia 001c.7390.93d3)
Hardware is Ethernet, address is 001c.7390.93d4 (bia 001c.7390.93d4)
Hardware is Ethernet, address is 001c.7390.93d5 (bia 001c.7390.93d5)
```

NOTE

Remember, Arista switches aren't similar to Linux devices; they *are* Linux devices. If what you're doing isn't working the way you'd expect it to, try thinking in terms of a Linux OS. More often than not, you'll find your answer in the way Linux works.

By the way, if you're like me and have been forever frustrated by the inability to get a simple list of the Ethernet interface's MAC addresses, check out this bit of Linux CLI/ Linux madness:

```
Arista(config)#show int | awk '/^[A-Z]/ {intf=$1} /, address
is/{print intf, $6}'
Ethernet1 001c.7390.93d0
Ethernet2 001c.7390.93d1
Ethernet3 001c.7390.93d2
Ethernet4 001c.7390.93d3
Ethernet5 001c.7390.93d4
Ethernet6 001c.7390.93d5
[--output truncated--]
```

If you think that's cool, save it as an alias:

```
Arista(config)#alias shmc show int | awk '/^[A-Z]/ {intf=$1} /,
address
is/{print intf, $6}'
Arista(config)#shmc
Ethernet1 001c.7390.93d0
Ethernet2 001c.7390.93d1
Ethernet3 001c.7390.93d2
Ethernet4 001c.7390.93d3
Ethernet5 001c.7390.93d4
Ethernet6 001c.7390.93d5
[--output truncated--]
```

Note that the alias command is one line and is wrapped in print to fit within the margins—it won't work if you enter it as two lines.

EOS runs a flavor of Linux, and many of the command behaviors

reflect that ancestry. For example, by default there is no pagination enabled in EOS when using the console. Therefore, when you execute the `show run` command, the output will scroll by until the entire running configuration has been displayed. To paginate on the fly, pipe your output to `more`. Just like using `more` in Linux, this will pause the output after the screen length (-1) has been met, at which point the prompt `--more--` will be shown. At this point, user input is required to continue. Pressing Enter or Return will result in the advancement of a single line, whereas pressing the space bar will show another page. Pressing the letter `q` (or `Q`) or pressing `Ctrl-C` will break the output and return you to the command prompt:

```
Arista-7280#sho run | more
! Command: show running-config
! device: Arista-7280 (DCS-7280SE-72, EOS-4.16.6F.idboise)
!
! boot system flash:/EOS-4.16.6M.swi
!
alias conint sh interface | i connected
alias senz show interface counter error | nz
alias snz show interface counter | nz
alias spd show port-channel %1 detail all
alias sqnz show interface counter queue | nz
alias srnz show interface counter rate | nz
!
alias intdesc
  !! Usage: intdesc interface-name description
  10 config
  20 int %1
  30 desc %2
  40 exit
!
transceiver qsfp default-mode 4x10G
--More--
```

Speaking of `show running-config`, there's a cool feature in EOS that will include all of the commands, even the defaults that aren't usually

shown. You do this by including the `all` keyword, and you can do it even when showing parts of the configuration such as an interface. Let me show you what I mean. Here's the output from the command `show run int e24`:

```
Arista#sho run int e24
interface Ethernet24
    switchport access vlan 901
    switchport mode trunk
```

And here's the output of the `show run all int e24` command, truncated for brevity. All of the commands you see here are active; they're just defaults, so they're not usually shown:

```
Arista-7280#sho run all int e24
interface Ethernet24
    description [ MLAG Peer ]
    no shutdown
    default load-interval
    logging event link-status use-global
    no dcbx mode
    no mac-address
    no link-debounce
    no flowcontrol send
    no flowcontrol receive
    no mac timestamp
    no speed
    no l2 mtu
    default logging event congestion-drops
[-- output truncated --]
```

Another nice function that you can invoke while piping is the `nz` command. This outputs only lines that have values of nonzero. Let's look at an example. Here, we have the output from the command `show interfaces counters errors`:

```
Arista-7280#sho int count error
Port      FCS      Align      Symbol      Rx      Runts      Giants      Tx
```

Et1	0	0	0	4	4	0	0
Et2	0	0	0	1	1	0	0
Et3	15	0	0	16	1	0	0
Et4	15	0	0	18	3	0	0
Et5	0	0	0	1	1	0	0
Et6	0	0	0	1	1	0	0
Et7	0	0	0	0	0	0	0
Et8	94	0	0	94	0	0	0
Et9	0	0	0	0	0	0	0
Et10	0	0	0	0	0	0	0
Et11	0	0	0	0	0	0	0
Et12	0	0	0	0	0	0	0
Et13	0	0	0	0	0	0	0
Et14	0	0	0	0	0	0	0
Et15	0	0	0	0	0	0	0
[-- output truncated --]							

Many of the lines have values of all zeros. Chances are, we don't care about this information, so why not display the same output without those lines? As here:

Arista-7280#sho interfaces counters errors nz							
Port	FCS	Align	Symbol	Rx	Runts	Giants	Tx
Et1	0	0	0	4	4	0	0
Et2	0	0	0	1	1	0	0
Et3	15	0	0	16	1	0	0
Et4	15	0	0	18	3	0	0
Et5	0	0	0	1	1	0	0
Et6	0	0	0	1	1	0	0
Et8	94	0	0	94	0	0	0
Et45	1	0	0	1	0	0	0

Notice that the lines that were all zero before are no longer included.

When I first started using Arista switches, I had been using Cisco Nexus switches for years, and the lack of the `| last` output modifier bugged me when using commands like `show log`. Then I sat and thought about what I knew about EOS and had a crazy idea. Remember earlier in this chapter when I showed how we could pipe to the Unix

grep command? It struck me that grep was not listed as one of the options when piping:

```
Arista#sho run | ?  
LINE      Filter command by common Linux tools such as grep/awk/sed/wc  
append    Append redirected output to URL  
begin     Begin with the line that matches  
exclude   Exclude lines that match  
include   Include lines that match  
json      Produce JSON output for this command  
no-more   Disable pagination for this command  
nz        Include only non-zero counters  
redirect  Redirect output to URL  
section   Include sections that match  
tee      Copy output to URL
```

I reasoned that if I could pipe to one Unix command, I would likely be able to pipe to others. The Unix command that would serve me here is `tail`, so I cast caution to the wind and went for it with all the vigor of a sleep-deprived nerd hell-bent on discovery.

Here's the output of the `show log` command piped through `more`:

```
Arista-7280#sho log | more  
Syslog logging: enabled  
  Buffer logging: level debugging  
  Console logging: level errors  
  Monitor logging: level errors  
  Synchronous logging: disabled  
  Trap logging: level informational  
  Sequence numbers: disabled  
  Syslog facility: local4  
  Hostname format: Hostname only  
  Repeat logging interval: disabled  
  
-----  
Facility          Severity      Effective Severity  
-----  
aaa              debugging    debugging  
accounting       debugging    debugging  
acl              debugging    debugging  
agent            debugging    debugging
```

ale	debugging	debugging
arp	debugging	debugging
bfd	debugging	debugging
bgp	debugging	debugging
capacity	debugging	debugging
capi	debugging	debugging
card	debugging	debugging
clear	debugging	debugging
cvx	debugging	debugging
dataplane	debugging	debugging
dot1x	debugging	debugging
envmon	debugging	debugging
eth	debugging	debugging
eventmon	debugging	debugging
--More--		

With bated breath, I entered my wicked conglomeration of EOS and Unix commands:

```
Arista-7280#sho log | tail
Sep 16 15:38:02 Arista-7280 Launcher: %LAUNCHER-6-PROCESS_START:
Configuring process 'FastClidHelper#2' to start in role
'AllSupervisors'
Sep 16 15:38:02 Arista-7280 Launcher: %LAUNCHER-6-PROCMGR_WARMSTART:
Initiating warm start of 'ProcMgr (worker)'
Sep 16 15:40:18 Arista-7280 Launcher: %LAUNCHER-6-PROCESS_STOP:
Configuring process 'FastClidHelper#2' to stop in role
'AllSupervisors'
Sep 16 15:40:18 Arista-7280 Launcher: %LAUNCHER-6-PROCMGR_WARMSTART:
Initiating warm start of 'ProcMgr (worker)'
Sep 16 15:43:05 Arista-7280 Cli: %SYS-5-CONFIG_I:
Configured from console by admin on con0 (0.0.0.0)
Sep 16 15:45:48 Arista-7280 Cli: %SYS-5-CONFIG_E: Enter
configuration mode from console by GAD on con0 (0.0.0.0)
Sep 16 15:48:33 Arista-7280 Rib: %OSPF-5-CONVERGED: OSPF in VRF
default has converged and its routes are in FIB
Sep 16 15:49:16 Arista-7280 Cli: %SYS-5-CONFIG_I: Configured from
console by GAD on con0 (0.0.0.0)
Sep 16 15:49:59 Arista-7280 Cli: %SYS-5-CONFIG_E: Enter configuration
mode from console by GAD on con0 (0.0.0.0)
Sep 16 15:55:29 Arista-7280 Cli: %SYS-5-CONFIG_I: Configured from
console by GAD on con0 (0.0.0.0)
```

Holy crap! It worked! Such was my enthusiasm that my wife came into

my home office at 2 a.m. and told me to go to bed, or at least keep it down. I suppose I should be lucky to have a wife, being that I'm a guy who finds the ability to mix EOS and Unix commands exciting.

As previously discussed, you can even stack Unix pipes:

```
Arista-7280#sho log | tail | grep GAD
Sep 16 15:45:48 Arista-7280 Cli: %SYS-5-CONFIG_E: Enter configuration
mode from
  console by GAD
on con0 (0.0.0.0)
Sep 16 15:49:16 Arista-7280 Cli: %SYS-5-CONFIG_I: Configured from
console by GAD on con0 (0.0.0.0)
Sep 16 15:49:59 Arista-7280 Cli: %SYS-5-CONFIG_E: Enter configuration
mode from console by GAD on con0 (0.0.0.0)
Sep 16 15:55:29 Arista-7280 Cli: %SYS-5-CONFIG_I: Configured from
console by GAD on con0 (0.0.0.0)
```

Why stop at two? Now that my wife had left me to my late-night spell casting, I ventured down the rabbit hole. How about adding some redirection?

```
Arista-7280#sho log | tail | grep GAD > /mnt/flash/GAD.txt
```

Because `tail` is a Unix command, we must specify the full path to get to the flash drive. You'll see what happens if you don't in [Chapter 12](#).

Now, my *GAD.txt* file is stored on the flash drive and can be seen with the `dir` command from EOS:

```
Arista-7280#dir
Directory of flash:/

-rwx 436644679      May 17 00:44  EOS-4.16.6M.swi
-rwx        27        Sep  6 23:27  boot-config
drwx      4096        Sep  6 23:34  debug
-rwx        424       Sep 16 16:09  gad.txt
```

```

-rwx        0          Aug 23 06:07  no_ssdd_var
drwx      4096        Sep  6 23:33  persist
drwx      4096        Aug 15 05:38  schedule
-rwx      1635        Sep  6 23:31  startup-config
-rwx        0          Sep  6 23:33  zerotouch-config

3547889664 bytes total (2658754560 bytes free)

```

Oh, when it comes to the log, there's a nifty capability called `show logging follow` that shows real-time updates to the syslog from the CLI. I can't really show that in a static book like this, but it behaves the same way as the `tail -f filename` command in Linux.

Lastly, there is a very cool option with `show running-config` that I really like. Have you ever wanted to see everything relating to Border Gateway Protocol (BGP)? Well, what if some of your BGP configuration isn't part of the router BGP section? Check out this overly simplified BGP configuration example:

```

Arista-7280(config)#ip routing
Arista-7280(config)#router bgp 65001
Arista-7280(config-router-bgp)#neighbor 10.0.0.100 remote-as 65100
Arista-7280(config-router-bgp)#comment
Enter TEXT message. Type 'EOF' on its own line to end.
GAD's Rockin BGP config
EOF
Arista-7280(config-router-bgp)#int e1
Arista-7280(config-if-Et1)#description BGP uplink
Arista-7280(config-if-Et1)#exit
Arista-7280(config)#

```

If I wanted to get all of the BGP commands in one fell swoop, how would I do it? I know, let's try piping to `inc`:

```

Arista-7280#sho run | inc bgp
router bgp 65001
Arista-7280#

```

Huh...well, that didn't work. What about all those lines with BGP in them? Maybe we need to search for BGP in all caps?

```
Arista-7280#sho run | inc BGP
  description BGP uplink
  !! GAD's Rockin BGP config
Arista-7280#
```

Now I have the opposite problem where I got only the uppercase BGP matches! **grep** to the rescue, right? Nope.

```
Arista-7280#sho run | grep BGP
  description BGP uplink
  !! GAD's Rockin BGP config
Arista-7280#
```

Wait a sec. If I think like a Linux guy, I know that I can use the **-i** flag with **grep** in order to ignore case:

```
Arista-7280#sho run | grep -i BGP
  description BGP uplink
  router bgp 65001
  !! GAD's Rockin BGP config
Arista-7280#
```

Hooray! No, wait...that's not good enough, either. Look at the description line. It doesn't show what interface that description is attached to. Luckily, EOS has a way to solve this problem using **section**. This works by using **show running-config | section <regex>**, where **<regex>** is a regular expression.

AMUSING DIGRESSION

If you don't know what a regular expression is, do yourself a favor and go buy the O'Reilly book *Mastering Regular Expressions* and read it. It will take you about six months if you're sharp. Go ahead, I'll wait...

With this powerful command, check out the output we receive:

```
Arista-7280#sho run | section bgp
interface Ethernet1
    description BGP uplink
!
router bgp 65001
    !! GAD's Rockin BGP config
    neighbor 10.0.0.100 remote-as 65100
    neighbor 10.0.0.100 maximum-routes 12000
Arista-7280#
```

Beautiful! Not only does it show every line that matches `bgp`, but it matches regardless of case and includes the entire section in which that match was found. This is a fabulous tool, and after you become accustomed to using it, you'll find that it's damn hard to live without. When troubleshooting a feature like MLAG, the command `show run | section mlag` is a powerful tool!

Conclusion

Although EOS has obvious similarities to the industry-standard CLI, there are also clear improvements and changes that should make any seasoned networking veteran happy. Not only that, but remember that the entire OS was written from scratch to be something new, so regardless of how things look, what's under the covers is arguably better than anything else out there.

Chapter 9. Configuration Management

In addition to the normal operations such as `write memory`, `copy run start`, and the like, the running configuration in Arista's Extensible Operating System (EOS) can be managed in a variety of interesting ways. To understand the majority of them, I'd like to begin with the command `configure replace`.

Configure Replace

Remember that Arista EOS, much like the industry standard CLI, is additive. In other words, if you want to remove a command from the running configuration, you must add a command to negate the existing command. For example, if I have a switch that has the following configuration

```
Arista(config-if-Et1)#sho active
  interface Ethernet1
    no switchport
    ip address 10.10.10.1/24
```

to remove the IP address, I must add a command to do so:

```
Arista(config-if-Et1)#no ip address
Arista(config-if-Et1)#sho active
  interface Ethernet1
    no switchport
```

Given this model, for decades the only way to load a completely new configuration on a device was either to carefully negate every existing command (a practical impossibility in most cases, not to mention a recipe for network disruption) or to overwrite the *startup-config* with something else and reload the device. This is a massively disruptive process that can take a large amount of time depending on the device and configuration in play.

In EOS 4.14, Arista added a new option to the `configure` command that allows you to replace the running configuration on a live switch. This is a very big deal that doesn't seem to be obvious to most of the people I show it to, and I think it's because the concept of wanting to replace the running configuration on a switch has never occurred to them, likely as a result of it simply not having been possible before now. As someone who maintains hundreds of switches in a lab and/or training environment, I can tell you that it's a tremendously powerful feature, especially when coupled with automation tools such as eAPI or CloudVision. In fact, if you are using CloudVision, every change to a switch done via CloudVision is being done through the use of `configure replace` (it's a touch more complicated than that, and I explain why later in this chapter). Consider this, though: during my classes I often change the entire behavior of a 24-switch lab using `config replace` automated through eAPI (Chapter 30). I can overwrite the *running-configs* of 24 switches in about 30 seconds, converting the entire thing from a Layer 2 (L2) Multi-Chassis Link Aggregation Group (MLAG) design to a Layer 3 (L3) Equal-Cost MultiPathing (ECMP) design without rebooting a single device and without disrupting connectivity to the switches. That's the power of

`config replace.`

I must point out that `config replace` does not do a *merge* operation. I had a heated discussion with someone in one of my classes in which the student insisted that what was happening was a merge operation, my guess being because she was used to other vendors' devices for which a real replace was not possible.

Configuration *merging* is the process by which any new commands are added but no commands will be removed. Let me illustrate the difference between merging and replacing with a simple configuration snippet:

```
Arista#sho run int e1
interface Ethernet1
  no switchport
  ip address 10.10.10.1/24
```

Here, I add a text file to *flash*: called *GAD.txt* that has the following contents:

```
Arista#more flash:GAD.txt
interface ethernet1
  description I Like Pie!
```

In EOS, copying a file into the *running-config* will merge the file into the *running-config*. Let's see what happens:

```
Arista#copy flash:GAD.txt running-config
! Missing final 'end' command, adding automatically.
Copy completed successfully.
```

```
Arista#sho run int e1
interface Ethernet1
  description I Like Pie!
```

```
no switchport
ip address 10.10.10.1/24
```

Sure enough, the contents of the *GAD.txt* file have been *added* to the *running-config* in a *merge* operation. Now let's take a look at why **config replace** is not the same thing. In this example, I reset the Ethernet configuration to be the same as it was when we started:

```
Arista#sho run int e1
interface Ethernet1
    no switchport
    ip address 10.10.10.1/24
```

Now, I do a **configure replace** using the same *GAD.txt* file:

```
Arista#configure replace flash:GAD.txt
! Missing final 'end' command, adding automatically.
```

Looks the same, but what you can't easily see reading a book is that my Secure Shell (SSH) connection died and the switch is completely offline. Why? Let's go into the console and find out:

```
localhost#sho run
! Command: show running-config
! device: localhost (DCS-7280SR-48C6-M, EOS-4.20.5F)
!
! boot system flash:/EOS-4.20.5F.swi
!
transceiver qsfp default-mode 4x10G
!
spanning-tree mode mstp
!
no aaa root
!
interface Ethernet1
    description I Like Pie!
!
interface Ethernet2
!
interface Ethernet3
```

```
!
interface Ethernet4
[-- output truncated --]
```

The hostname is no longer Arista and has changed to `localhost`, which is the default hostname for an Arista switch. If you look at the configuration for Ethernet1, you can see that it contains the contents of the `GAD.txt` file but *not* the original configuration. So why did this happen? Because `config replace` does not do a *merge* operation. Instead, `config replace` does what the name implies—it *replaces* the running configuration with the contents of the source, which in this case was `flash:GAD.txt`.

By the way, check out how simple it is to fix my catastrophic blunder using `config replace` again to literally replace the *running-config* with the contents of the *startup-config*.

```
localhost#configure replace flash:startup-config
localhost#<cr>
Arista#
```

After the replace is finished, I need to press Return, but then I’m right back where I started. Understand that at no time did I reboot this switch! I completely replaced the running configuration on a live switch. Sure, I caused a network outage with the original `config replace`, but that’s not important right now.

Now, if you’re wondering whether that could be bad (and you weren’t convinced when I wiped out the entire configuration of a live switch), take heart in that there’s more to `config replace` than wiping out the *running-config*. What’s especially nice about this feature is that if a

section remains unchanged, there is no disruption to that process. In other words, if I do a `config replace` and both the existing *running-config* and the new one have the exact same 30 lines of Border Gateway Protocol (BGP) configuration, there will be no interruption to BGP on the switch (assuming IPs and such also remain unchanged). And the same is true for any configuration section: spanning-tree, mlag, eapi, vxlan—it doesn't matter.

So why then, did my previous example result in the switch being wiped out? Because I replaced the entire *running-config* with a file containing only these lines:

```
Arista#more flash:GAD.txt
interface ethernet1
    description I Like Pie!
```

I can't say this often enough: `config replace` replaces the *running-config*! Because I replaced the entire file with only two lines, everything else that had been configured previously reverted to the default, and that's why I lost connectivity and the hostname defaulted to `localhost`. So what's the solution?

Don't do that.

When using `config replace`, you should always think about replacing the entire configuration, because that's what it's for. If what you want is a merge, use `copy source running-config`. I know I've harped on that point, but what I'm going to cover next builds on the assumption that you understand `config replace`.

Configuration Checkpoints

EOS also has the ability to save configuration checkpoints. This feature allows you to save the running configuration as it exists at a point in time so that you can return to it at some point in the future. To save a checkpoint use the `configure checkpoint save checkpoint-name` command:

```
Arista(config)#config checkpoint save GAD-Pre-Change
```

With one or more checkpoints saved you can view a brief report of them using the `show config checkpoints` command:

```
Arista(config)#sho config checkpoints
Maximum number of checkpoints: 20
  Filename          Date        User
  -----
  GAD-Pre-Change    2019-01-02 22:30:43  admin
```

With a checkpoint saved, it can now be restored using the `checkpoint restore checkpoint-name` command, but before we do that, I need to show you a limitation of how this all works that might not be immediately obvious. Actually, it's not a limitation at all, but in my experience, people seem to want to believe that it works differently, so I want to prevent problems. You see, configuration checkpoints work through the ability of EOS to do a `config replace`. That means that the checkpoint, when restored, *replaces* the current *running-config*. This is not a merge operation! Allow me to prove my point.

Remember that we just saved a configuration checkpoint named **GAD-Pre-Change**. After that save, I'm going to go in and change the *running-config* through good old-fashioned CLI commands:

```
Arista(config)#conf
Arista(config)#int e5
Arista(config-if-Et5)#sho active
interface Ethernet5
```

At this point we can see that's there is no configuration on Ethernet5, so let's add a description:

```
Arista(config-if-Et5)#desc I Like Pie!
Arista(config-if-Et5)#sho active
interface Ethernet5
description I Like Pie!
```

Now Ethernet5 has a description configured on it. I know, this is not exactly a BGP configuration for a major internet peering point, but stay with me. I'm now going to restore the checkpoint that I saved earlier:

```
Arista(config-if-Et5)#configure checkpoint restore GAD-Pre-Change
```

Now that I've done that, let's see what the configuration on E5 looks like:

```
Arista(config-if-Et5)#sho active
interface Ethernet5
```

And just like that, my interface description went away. Why? Because the configuration checkpoint feature uses `config replace` to restore a checkpoint, and (can you guess what's coming?) `config replace` *does not do a merge*.

You might have noticed before when I listed the checkpoints that there was a maximum number of checkpoints listed:

```
Arista(config)#sho config checkpoints
Maximum number of checkpoints: 20
```

Filename	Date	User
GAD-Pre-Change	2019-01-02 22:30:43	admin

You can configure this number via the **service configuration checkpoint max saved *number*** command:

```
Arista(config)#service configuration checkpoint max saved ?
<0-100> Maximum number of saved checkpoints
Arista(config)#service configuration checkpoint max saved 50
```

Let's save another and then see what the report looks like:

```
Arista(config)#sho configuration checkpoints
Maximum number of checkpoints: 50
-----
```

Filename	Date	User
GAD-Post-Change	2019-01-02 22:30:43	admin
GAD-Pre-Change	2019-01-02 22:30:34	admin

If you're wondering where these checkpoints are stored, it's in a hidden directory on *flash*: named *.checkpoints*:

```
[admin@Arista ~]$ ls -al /mnt/flash/.checkpoints
total 16
drwxrwx--- 2 root eosadmin 4096 Jan  2 22:30 .
drwxrwx--- 7 root eosadmin 4096 Jan  2 22:09 ..
-rwxrwx--- 1 root eosadmin 2928 Jan  2 22:30 GAD-Post-Change
-rwxrwx--- 1 root eosadmin 2928 Jan  2 22:30 GAD-Pre-Change
```

That means that checkpoints will survive a reboot. It also means that they take up space on *flash*. The fact that flash drive space is at a premium on many switches is likely why the default is to store only 20 checkpoints. Because each of these files is clear text (they are not zipped) and some devices have very large configurations, be cognizant of the fact that your checkpoints are consuming space on *flash*—space

that is not reported in the output of the `show config checkpoints` CLI command.

As a fun aside (I think it's fun, but then I've been told that I'm not exactly normal), you can do a `config replace` directly from a checkpoint file if the mood strikes you because there is now a device in CLI called `checkpoint:`. This makes perfect sense because the checkpoint files are nothing more than text files.

```
Arista(config)#configure replace checkpoint:GAD-Pre-Change
```

I should point out that there's really no reason to do this because you can accomplish the same thing using `config checkpoint restore`, but I like the fact that this works because it shows how the Arista developers work to make things behave the way they should. Actually, I'd be willing to bet that this is just a logical by-product of building it the right way from the ground up.

Configuration Sessions

We've looked at `config replace` and configuration checkpoints, so the last thing we're going to cover in this chapter is configuration *sessions*. Configuration sessions allow for a couple of interesting options that more or less come down to the idea that a group of commands can be saved and applied at a point in time instead of interactively through the CLI. Again, though, this ability derives from the root idea of `config replace`, so this might not work the way you expect if you're used to merging. It certainly caught me by surprise, and I'm the guy who's been ranting and gesticulating about these

features not doing a merge. Guess what? Configuration sessions do not do a merge, either. Let's take a look.

The principal idea of **config session** is that you enter a bunch of CLI commands but those commands are not applied when you enter them. Instead, they are saved for later use. Actually, that's not strictly true, which is what leads to the confusion. Let's see why.

First, let's begin with an empty interface configuration:

```
Arista(config)#sho run int e5
interface Ethernet5
```

I then start a new **config session** named GAD and put in the configuration commands that I would use to save the configuration:

```
Arista#conf session GAD
Arista(config-s-GAD)#int e5
Arista(config-s-GAD-if-Et5)#desc I Like Pie!
Arista(config-s-GAD-if-Et5)#^z
Arista#
```

This string of commands has not changed the *running-config*, but instead has saved a new **config session** named GAD. With a session saved, I can view a report of all sessions using the **show config sessions** command:

```
Arista#show config session
Maximum number of completed sessions: 1
Maximum number of pending sessions: 5

      Name      State      User      Terminal
      -----  -----
      GAD      pending
```

What tends to get people is that they think that the session is nothing more than the couple of commands typed to create the session, but that is not the case. Using the command `show session-config named session-name` shows the contents of a session. Here's the output for my GAD session:

```
Arista#show session-config named GAD
! Command: show session-configuration named GAD
! device: Arista (DCS-7280SR-48C6-M, EOS-4.21.1F)
!
! boot system flash:/EOS-4.21.1F.swi
!
alias conint sh interface | i connected
alias senz show interface counter error | nz
alias shmc show int | awk '/^[A-Z]/ {intf = $1}/, address is/{print intf,
$6}'
alias snz show interface counter | nz
alias spd show port-channel %1 detail all
alias sqnz show interface counter queue | nz
alias srnz show interface counter rate | nz
[--output truncated--]
```

Holy moly—that's a *running-config*! Why? Because that's how configuration sessions work. They don't save the commands you entered; rather, they make a copy of the *running-config* and apply those commands to *that*, thus forming a new configuration with your changes added, which is then saved as a session.

WARNING

Let me reiterate that. A configuration session is not a set of commands you entered. It is the *running-config* at the time you entered them, merged with your commands. The result is a whole new configuration—not just a list of the commands you entered.

To prove this further, you can use the `show session-config named`

session-name **diff** command:

```
Arista#sho session-config named GAD diff
--- system:/running-config
+++ session:/GAD-session-config
@@ -43,6 +43,7 @@
  interface Ethernet4
 !
  interface Ethernet5
+  description I Like Pie!
 !
  interface Ethernet6
 !
```

Without getting deep into how **diff** works, this output is showing that the only difference between the *running-config* and the session named **GAD** is the one line, **description I Like Pie!**

Now, without applying that session, I'm going to change the *running-config* for Ethernet 6:

```
Arista#conf
Arista(config)#int e6
Arista(config-if-Et6)#desc I hate cake
Arista(config-if-Et6)#^z
```

At this point I have changed the *running-config* (specifically for E6), but I did not alter the *config-session* (which changed E5). Now when I do a **show session-config named GAD diff** to show the differences between the *running-config* and the session named **GAD**, notice that the **GAD** session does not include a description on int e6:

```
Arista#
Arista#sho session-config named GAD diff
--- system:/running-config
+++ session:/GAD-session-config
@@ -43,9 +43,9 @@
```

```
interface Ethernet4
!
interface Ethernet5
+  description I Like Pie!
!
interface Ethernet6
-  description I hate cake
!
interface Ethernet7
!
```

If the **diff** output is confusing, how about I just commit the session and show what happens? Here is the current *running-config* for Ethernet 5 and 6:

```
Arista#sho run int e5-6
interface Ethernet5
interface Ethernet6
  description I hate cake
```

Now I commit the session named **GAD** that has the description for E5. Configuration sessions are applied by using the **commit** command from within configuration mode for the session that you want to commit:

```
Arista#config session GAD
Arista(config-s-GAD)#commit
```

Let's see what happened to our configurations for E5–6:

```
Arista#sho run int e5-6
interface Ethernet5
  description I Like Pie!
interface Ethernet6
```

While the configuration for Ethernet5 was added, note that the configuration for Ethernet6 was removed. Why? Because the session *replaced the running-config* with what was contained in the session.

WARNING

Configuration sessions do not contain just the commands you entered; they contain a *running-config* from the time the session was created with those commands applied. If you make changes to the *running-config* and then apply the session, you will lose the changes you made to the *running-config*.

I should point out that there are a couple of cool things about configuration sessions that I haven't covered yet. First, you can commit a session with a timer.

```
Arista#configure session GAD2
Arista(config-s-GAD2)#int e7
Arista(config-s-GAD2-if-Et7)#description I like pudding
Arista(config-s-GAD2-if-Et7)#^z
```

This time, I commit the change, but with a timer:

```
Arista#configure session GAD2
Arista(config-s-GAD2)#commit timer 00:03:00
```

What this does is commit the change, but the change will stay only for the duration of the timer unless someone goes in and commits without a timer. Why would you do this? To prevent you from being locked out of the switch when issuing remote commands. I once shut down all of Australia for a global financial firm by applying an empty Access Control List (ACL) to a router (whoops!), which is when I learned about the `reload in time` command. Using a commit timer is much more powerful than the `reload in` command because there's no reboot involved. Note that the command I entered does not commit the change *in* three minutes: it commits the change *for* three minutes. Here

is the *running-config* for E7 after I issued the command:

```
Arista#sho run int e7
interface Ethernet7
  description I like pudding
```

While the session is committed and the timer is active, you can see the time remaining by using the **show config session detail** command:

```
Arista#sho configuration sessions detail
Maximum number of completed sessions: 1
Maximum number of pending sessions: 5

  Name  State          User      Terminal      PID      Commit
Time Left
  -----
  -----
  GAD   completed
  GAD2  pendingCommitTimer
1m30s
```

When the timer expires and the session has not been permanently committed, the system reverts to a checkpoint that the system made automatically at the time of the timed commit:

```
Arista#sho configuration sessions detail
Maximum number of completed sessions: 1
Maximum number of pending sessions: 5

  Name  State          User      Terminal      PID      Commit
Time Left
  -----
  -----
  cfg-2625--896533696-0  completed

  Name          Description
  -----
  cfg-2625--896533696-0 config replace of commitTimerCheckPointConfig
```

Unlike with configuration checkpoints, configuration sessions are not saved to *flash*: and will not survive a reboot, so don't expect to be able to make a bunch and use them post-upgrade.

NOTE

Technically, `configure replace` exists as a result of the structure built for `config sessions`, but I explain them in a different order because it seems easier to explain the idea as building on the idea of replacing a configuration.

As a quick final note, if you've wondered why the output of `show config session` doesn't show a user and terminal in my examples, it's because those fields are populated only when a user is actively in a session.

```
Arista#configure session GAD
Arista(config-s-GAD)#sho configuration sessions
Maximum number of completed sessions: 1
Maximum number of pending sessions: 5

      Name      State      User      Terminal
      -----  -----
* GAD      pending    admin      vty3
```

As a last bit of interesting information regarding `config session`, you can actually make a default `config session` using the `rollback clean-config` command from within a session:

```
Arista#configure session GAD
Arista(config-s-GAD)#rollback clean-config
Arista(config-s-GAD)#exi
```

Doing this will create a configuration that has only default commands,

which will give the same results as doing a `write erase` and then a `reload`.

```
Arista#sho session-config named GAD
! Command: show session-configuration named GAD
! device: Arista (DCS-7280SR-48C6-M, EOS-4.21.2F)
!
! boot system flash:/EOS-4.21.2F.swi
!
transceiver qsfp default-mode 4x10G
!
spanning-tree mode mstp
!
no aaa root
!
interface Ethernet1
!
interface Ethernet2
!
[-- snip--]
!
interface Ethernet47
!
interface Ethernet48
!
interface Ethernet49/1
!
interface Ethernet50/1
!
interface Ethernet51/1
!
interface Ethernet52/1
!
interface Ethernet53/1
!
interface Ethernet54/1
!
interface Management1
!
no ip routing
!
end
```

I'm not sure how useful this is in a production environment, but in a

lab environment, where you might need to have a “blank” or default switch, this is a pretty powerful capability and is actually part of how CloudVision uses `config session` to configure switches on live networks.

EOS limits the number of sessions to five by default, but you can change this by using the `service configuration session max` command:

```
Arista(config)#service configuration session max ?
completed  maximum number of completed sessions kept in memory
pending    maximum number of pending sessions

Arista(config)#service configuration session max completed ?
<0-20>  Maximum number of saved sessions

Arista(config)#service configuration session max pending ?
<1-20>  Maximum number of pending sessions
```

Because it would appear that the configuration sessions reside in memory and are not compressed and because some switches have some large configurations, I probably wouldn’t change these values unless I really needed to.

Conclusion

Configuration management in EOS is very powerful, but remember that it might not work the way you expect it to if you’re familiar with the way that other networking vendors deal with the same topic. When in doubt, remember that it’s not a merge unless you use the `copy` command. Also, testing this out for yourself in a lab environment would probably be a good idea before you go replacing the running

configuration on a production device. Or not. I suppose there are some people who actually enjoy job hunting.

Chapter 10. Upgrading EOS

Upgrading Arista's Extensible Operating System (EOS) is a simple process. In a nutshell, get the software image onto the switch, configure the switch to see the new image, and then reboot the switch. Let's begin by taking a look at the current version of EOS on a 7280R-48C6 switch by using the `show version` command:

```
Arista#sho ver
Arista DCS-7280SR-48C6-M-F
Hardware version: 21.05
Serial number: SSJ17290598
System MAC address: 2899.3abe.9f92

Software image version: 4.19.5M
Architecture: i386
Internal build version: 4.19.5M-7506183.4195M
Internal build ID: ef83948c-6c45-4097-9d8b-a78a4722d49a

Uptime: 3 minutes
Total memory: 32463792 kB
Free memory: 30178840 kB
```

We can see from this output that our switch is running EOS version 4.19.5M.

NOTE

Once again, check out the model number in the first line of output, which reads DCS-7280SR-48C6-M-F. The “-F” indicates that this switch shipped with front-to-rear airflow fans. If the model number had “-R” appended to it, the fans would be rear-to-front. Cool, huh? Well, I think it is, but then, little things like this excite me. The “-M” indicates that this switch has expanded memory (32 GB). The “SR” means that this is an Small Form-Factor Pluggable (SFP)-based (S) switch with advanced routing (R) capabilities. The “48C6”

means there are 48 front-panel SFP interfaces along with six 100 Gb (C) interfaces.

If we felt like poking around further where we don't belong (and who doesn't?), we could use the `show version detail` command, which would regale us with page upon page of process names and their requisite versions:

```
Arista#sho ver detail
Arista DCS-7280SR-48C6-M-F
Hardware version: 21.05
Deviations: D0003570, D0003550
Serial number: SSJ17290598
System MAC address: 2899.3abe.9f92

Software image version: 4.19.5M
Architecture: i386
Internal build version: 4.19.5M-7506183.4195M
Internal build ID: ef83948c-6c45-4097-9d8b-a78a4722d49a

Uptime: 7 minutes
Total memory: 32463792 kB
Free memory: 29957796 kB
```

Installed software packages:

Package	Version	Release
Aaa	1.1.1	7498111.4195M
Aaa-cli	1.1.1	7498111.4195M
Aaa-lib	1.1.1	7498111.4195M
Aboot-utils	1.0.2	7498111.4195M
Acl	1.0.2	7498111.4195M
Acl-cli	1.0.2	7498111.4195M
Acl-lib	1.0.2	7498111.4195M
AclAgent	1.0.2	7498111.4195M
AclAgent-cli	1.0.2	7498111.4195M
AclAgent-lib	1.0.2	7498111.4195M

[-- pages of output removed to keep my editor from hurting me --]

zile	2.4.9	1.fc18
------	-------	--------

zip	3.0	5.fc18
zlib	1.2.7	9.fc18
Component	Version	
<hr/>		
Aboot	Aboot-norcal6-6.1.2-4757975	
scd	0xd	
stdbycpld	0x15	
Ucd9012-2	SFT004190111 (2.3.4.0011)	
Ucd9012-3	SFT004100105 (2.3.4.0007)	
System Epoch:	02.00	

NOTE

See the line in bold (as well as many others) in the previous output that ends with **.fc18**? This indicates that the package on that line was built on Fedora Core 18.

Interesting information if you're Arista TAC (Technical Assistance Center), but probably overkill for us. Our goal in this chapter is to upgrade this switch from EOS version 4.19.5M to version 4.21.1F (which, by the way, is built on Fedora Core 18). So, let's get started.

The first thing to do is take a look at what devices there are available to us on this Arista switch by using the **dir ?** command:

```
Arista#dir ?
  all/filesystems  List files on all filesystems
  certificate:    Directory or file name
  drive:          Directory or file name
  extension:      Directory or file name
  file:           Directory or file name
  flash:          Directory or file name
  sslkey:         Directory or file name
  system:         Directory or file name
  /all            List all files, including hidden files
  /recursive      List files recursively
  <cr>
```

Arista switches all contain useful USB ports. [Figure 10-1](#) shows the USB port on 7280R switch, located on the back of the device (there is also one on the front of this model). After years of using switches from other vendors that contained USB ports that were unavailable, being able to slap a thumb drive in and actually make use of it came as a pleasant surprise.

Upon insertion of a USB drive, the `dir ?` command yields different results:

```
Arista-7280#dir ?
  all-filesystems  List files on all filesystems
  certificate:    Directory or file name
  drive:          Directory or file name
  extension:      Directory or file name
  file:           Directory or file name
  flash:          Directory or file name
  sslkey:         Directory or file name
  system:         Directory or file name
  usb1:           Directory or file name
  /all            List all files, including hidden files
  /recursive      List files recursively
<cr>
```



Figure 10-1. The management, console, and usb2: ports on the back of an Arista 7280R

Now we see the `usb1:` device, which was not there previously. Note also that this switch has a `drive:` device. That indicates that this switch was shipped with a solid-state drive (SSD) drive, which we can see by using the `show inventory` command:

```
Arista#sho inventory | egrep -A4 "has SSD"
System has SSD
  Model           Serial Number     Rev      Size (GB)
  -----
  StorFly VSFBM8CC120G-160 P1T13004414108250019 0728-000 120
```

Executing the `dir` command by itself will default to the current directory, which is in `flash:/`:

```
Arista#dir
Directory of flash:/

  -rwx  609823300          Feb  8  2018  EOS-4.19.5M.swi
  -rwx      77            Jan  3 22:17  SsuRestore.log
  -rwx      27            Jan  3 22:15  boot-config
  drwx    4096            Jan  3 22:19  debug
  drwx    4096            Jan  3 22:19  persist
  drwx    4096            Jan  2 22:09  schedule
  -rwx    2983            Jan  3 22:15  startup-config
  -rwx        0            Jan  3 22:18  zerotouch-config

3269361664 bytes total (2047406080 bytes free)
```

As we can see, there is an existing EOS image in this directory called `EOS-4.19.5M.swi`. This image corresponds to the version shown earlier with the `show version` command. The `M` in the version is a reference to where this EOS release sits in the life cycle of the release, with “M” indicating a *Maintenance* release and “F” indication a *Feature* release. You can read the [EOS life cycle policy at Arista.com](#).

You can also find references to the image file in the running

configuration. We use the command found here to upgrade the system in this chapter:

```
Arista#sho run | grep boot
! boot system flash:/EOS-4.19.5M.swi
```

If you have a sharp eye, you might have noticed that the command is commented out. This is due to the way that EOS actually operates. This will become apparent later in this chapter.

One of the cool things about EOS is that it allows a wide variety of ways to get new files onto the switch. Check out the possibilities included in the `copy ?` output:

```
Arista#copy ?
  boot-extensions      Copy boot extensions configuration
  certificate:         Source file path
  clean-config         Copy from clean, default, configuration
  drive:               Source file path
  extension:           Source file path
  file:                Source file path
  flash:               Source file path
  ftp:                 Source file path
  http:                Source file path
  https:               Source file path
  installed-extensions Copy installed extensions status
  running-config       Copy from current system configuration
  scp:                 Source file path
  sftp:                Source file path
  startup-config       Copy from startup configuration
  system:              Source file path
  terminal:            Source file path
  tftp:                Source file path
```

To do this, we use the `copy source destination` command. Arista allows a variety of source and destinations when copying.

In this example, we're going to copy from a web page within a private

lab, so the source will be `http://url/filename`, and the destination `flash:`. If we were to source the file from a TFTP server, the source would be `tftp`.

```
Arista#copy http://10.0.0.100/EOS/EOS-4.21.1F.swi flash:  
'/EOS/EOS-4.21.1F.swi' at 469575994 (66%) 101.87M/s eta:2s [Receiving  
data]
```

When the copy is done, we can verify that it resides in its new home in the `flash:/` directory.

```
Arista#dir  
Directory of flash:/  
  
-rwx 609823300 Feb 8 2018 EOS-4.19.5M.swi  
-rwx 700978970 Nov 12 2018 EOS-4.21.1F.swi  
-rwx 77 Jan 3 22:17 SsuRestore.log  
-rwx 27 Jan 3 22:15 boot-config  
drwx 4096 Jan 3 22:19 debug  
drwx 4096 Jan 3 22:19 persist  
drwx 4096 Jan 2 22:09 schedule  
-rwx 2983 Jan 3 22:15 startup-config  
-rwx 0 Jan 3 22:18 zerotouch-config  
  
3269361664 bytes total (1346424832 bytes free)
```

Now that we have our EOS image safely on board, we can configure the system to use it. Note that EOS will *not* simply boot the first image it finds on flash. The reason for this will become apparent when we discuss Aboot, but for now let's go ahead and configure the system to boot from the new image. As with other vendors, the command to do this is the `boot system` command:

```
Arista#conf  
Arista(config)#boot system ?  
drive: Software image URL  
file: Software image URL  
flash: Software image URL
```

If we had a USB stick in the USB slot, the `usb1:` filesystem would be included in the output. We could configure the switch to boot from an image on the USB drive if we so desired. Almost anything we can do with normal flash, we can also do with USB, provided that there is a drive in the USB slot. For now, though, let's go ahead and configure the switch to boot from its new image in flash:

```
Arista(config)#boot system flash:EOS-4.21.1F.swi
Arista(config)#
```

EOS supports tab completion, so instead of copying and pasting the entire filename, try using the Tab key. Luckily, Arista keeps all of the image names short and tidy, which is another pleasant surprise after decades of other vendors' lengthy filenames.

Now that we have the new image primed and ready to go, let's reboot the switch:

```
Arista-7280(config)#reload
Proceed with reload? [confirm]
```

Hold on a minute! We didn't save the configuration, so why didn't the system prompt us to save it before rebooting? The `boot system` command doesn't really make changes to the *running-config*. Instead, it writes to a file on *flash*: called *boot-config*:

```
Arista(config)#dir
Directory of flash:/

-rwx 609823300          Feb  8 2018  EOS-4.19.5M.swi
-rwx 700978970          Nov 12 2018  EOS-4.21.1F.swi
-rwx          77          Jan  3 22:17  SsuRestore.log
-rwx          27          Jan  3 22:38  boot-config
drwx        4096          Jan  3 22:19  debug
```

```

drwx        4096          Jan 3 22:19  persist
drwx        4096          Jan 2 22:09  schedule
-rwx        2935          Jan 3 22:38  startup-config
-rwx          0           Jan 3 22:18  zerotouch-config

3269361664 bytes total (1346424832 bytes free)

```

Boot-config is the filename used for something called *Aboot*. *Aboot* is the bootloader for the switch (which is really a Linux system, remember), which I cover in more detail in [Chapter 14](#). For now, understand that without the bootloader's configuration file, our switch will not boot. You can view the contents of the *boot-config* file in a couple of ways, the easiest of which is by using the `show boot` command.

```

Arista(config)#sho boot
Software image: flash:/EOS-4.21.1F.swi
Console speed: (not set)
Aboot password (encrypted): (not set)
Memory test iterations: (not set)

```

In the first edition of *Arista Warrior*, I took the liberty of deleting this file to prove my point. With both image files present and no *boot-config* file found, the switch floundered shortly after initialization:

```

Arista(config)#reload
System configuration has been modified. Save? [yes/no/cancel/diff]:y
Proceed with reload? [confirm]

Broadcast meStopping sshd: [ OK ]
[ 575.544430] SysRq : Remount R/O
Restarting system

Aboot 1.9.2-140514.2006.eswierk
Press Control-C now to enter Aboot shell
No SWI specified in /mnt/flash/boot-config
Welcome to Aboot.
Aboot#

```

At this point, the switch was dead in the water. What's worse, if you've ever been near a modern switch from most any vendor when it boots, you'll know that all of the fans go to full power upon power up and then slow down to an acceptable level after boot up. With no EOS image to load, the switch spins its fans at full speed. It's annoying, and no one likes it, especially my wife, who came down to yell at me about all the noise while I was writing this chapter. But I digress...

On modern versions of code, the switch will actually warn us if there is no *boot-config* file:

```
Arista(config)#del flash:boot-config
Arista(config)#reload
The boot image ((not set)) is not present.
Are you sure you want to reload? [confirm]
```

Let's fix our boot image problem by using the `boot system` command:

```
Arista(config)#boot system flash:EOS-4.21.1F.swi
Arista(config)#
```

From this point on, the switch will load the specified image on reload. Here's the final output of `show version` before we reboot with the new code:

```
Arista(config)#sho ver
Arista DCS-7280SR-48C6-M-F
Hardware version: 21.05
Serial number: SSJ17290598
System MAC address: 2899.3abe.9f92

Software image version: 4.19.5M
Architecture: i386
Internal build version: 4.19.5M-7506183.4195M
Internal build ID: ef83948c-6c45-4097-9d8b-a78a4722d49a
```

Uptime:	26 minutes
Total memory:	32463792 kB
Free memory:	29917256 kB

Note that sometimes when you upgrade a switch, you'll see one or more messages like this, which indicates that new microcode is being loaded onto the Field Programmable Gate Arrays (FPGA), which are essentially programmable Application-Specific Integrated Circuits (ASICs):

```
-----  
Upgrading the casini system fpga.  
This process can take 2 minutes.  
Please do not reboot your switch.
```

```
-----  
Upgrade of the casini system fpga completed successfully.
```

Without the diversion for showing you how the switch reacts to not having a *boot-config* file, the process was very simple: get the code on the system, update the *boot-config* file with the `boot system` command, and reload.

Finally, here's the output of `show version` after the upgrade:

```
Arista#sho ver  
Arista DCS-7280SR-48C6-M-F  
Hardware version: 21.05  
Serial number: SSJ17290598  
System MAC address: 2899.3abe.9f92  
  
Software image version: 4.21.1F  
Architecture: i386  
Internal build version: 4.21.1F-9887494.4211F  
Internal build ID: 1497e24b-a79b-48e7-a876-43061e109b92  
  
Uptime: 0 weeks, 0 days, 0 hours and 6 minutes  
Total memory: 32458980 kB  
Free memory: 30458056 kB
```

If you're running Multi-Chassis Link Aggregation Group (MLAG), check out [Chapter 18](#) for details about upgrading using MLAG In-Service Software Upgrade (ISSU), which allows for a switch pair to be upgraded with minimal packet loss and no STP reconvergence!

Conclusion

As someone who's been in the field for far too long, I've installed or upgraded thousands of switches in my lifetime, and even after being at Arista for more than six years, I'm still impressed at the ease with which I can change the code revision on Arista devices. With automation tools like CloudVision ([Chapter 15](#)) or eAPI ([Chapter 30](#)), it becomes even easier. Be warned, though, that it's so simple that you'll likely grow annoyed when you have to go back to upgrading other vendors' devices!

Chapter 12. Bash

Arista switches are really Linux servers optimized and programmed to be network switches. By this point in the book, that should not be a surprise, but what might be surprising is the depth to which you, the administrator, can gain access to the system.

NOTE

If you really don't like the idea of junior engineers having access to Bash, you can limit their access to it by using AAA.

To access Bash, type the command `bash` from the enable prompt:

```
Arista-7280#bash
Arista Networks EOS shell
[admin@Arista-7280 ~]$
```

At this point, I am within a Bash shell on the switch. The prompt, by default, will be `[username@hostname directory]$`. In the previous example, I logged in to the switch with the default username (admin). I have not created a username in Unix; the switch took care of that for me.

At this point, I have just about all the control that I would have as a user in Linux. I am not a *superuser*, and my home directory is empty:

```
[admin@Arista-7280 ~]$ ls
[admin@Arista-7280 ~]$
```

I can navigate around the filesystem, just like I can on a Linux server:

```
[admin@Arista-7280 ~]$ cd /
[admin@Arista-7280 /]$ cd /usr/
[admin@Arista-7280 usr]$ ls
bin  etc  games  include  lib  libexec  local  sbin  share  src  tmp
[admin@Arista-7280 usr]$
```

If you're at all familiar with Linux, you'll be right at home in this Bash shell:

```
[admin@Arista-7280 usr]$ ls -alh
total 0
drwxr-xr-x 16 root root 120 Oct 29 21:28 .
drwxr-xr-x 29 root root 320 Nov 14 19:33 ..
dr-xr-xr-x  2 root root 880 Nov 14 19:34 bin
drwxr-xr-x  2 root root   3 Jul 19 2012 etc
drwxr-xr-x  2 root root   3 Jul 19 2012 games
drwxr-xr-x  5 root root 101 Oct 29 21:28 include
dr-xr-xr-x 77 root root 5.1K Nov 14 19:33 lib
drwxr-xr-x 10 root root 551 Oct 29 21:30 libexec
drwxr-xr-x 11 root root 127 Oct 29 21:28 local
dr-xr-xr-x  2 root root   60 Nov 14 19:33 sbin
drwxr-xr-x 85 root root 220 Oct 29 21:30 share
drwxr-xr-x  4 root root   43 Oct 29 21:28 src
lrwxrwxrwx  1 root root   10 Nov  2 07:10 tmp -> ../../var/tmp
[admin@Arista-7280 usr]$
```

To prove the point that an Arista switch is a Linux server with specialized interface hardware, I'll show the network interfaces from Bash:

```
[admin@Arista-7280 ~]$ ifconfig -a | more
cpu: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 9216
      ether 00:1c:73:90:93:cf  txqueuelen 1000  (Ethernet)
      RX packets 0  bytes 0 (0.0 B)
      RX errors 0  dropped 0  overruns 0  frame 0
      TX packets 0  bytes 0 (0.0 B)
```

```

TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

cpudebug: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 9216
      ether 00:1c:73:90:93:cf txqueuelen 1000 (Ethernet)
      RX packets 0 bytes 0 (0.0 B)
      RX errors 0 dropped 0 overruns 0 frame 0
      TX packets 0 bytes 0 (0.0 B)
      TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

et1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 9214
      ether 00:1c:73:90:93:cf txqueuelen 1000 (Ethernet)
      RX packets 0 bytes 0 (0.0 B)
      RX errors 0 dropped 0 overruns 0 frame 0
      TX packets 504 bytes 64328 (62.8 KiB)
      TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

[-- output truncated --]

```

Heck, even **vmstat** works:

```

[admin@Arista-7280 ~]$ vmstat 1 5
procs -----memory----- -swap- ---io-- -system-- -----cpu-----
  r  b  swpd   free   buff   cache   si  so   bi   bo   in   cs   us   sy   id   wa
  0  0     0 119916 225664 1820532   0  0 232  25  420  385 14  3 83  1
  0  0     0 119908 225664 1820532   0  0   0   0 1294  605  1  0 99  0
  0  0     0 119908 225664 1820532   0  0   0   0 1940 2115  4  1 95  0
  0  0     0 119908 225664 1820532   0  0   0   0 1617 1316  2  0 98  0
  0  0     0 119908 225664 1820532   0  0   0   0 1435  884  2  0 98  0

```

I feel it is important to reiterate that all of these Linux commands work because the Arista switch is a Linux machine. This is not a bash emulation; this is bash. It is more accurate to think that the command-line interface (CLI) on the Arista switch is a switch OS emulation; although to be painfully accurate, that is not right, either.

The CLI environment on an Arista switch is a process in Linux. We can see this from Bash by executing the command **Cli**. Here, I spawn a CLI session, execute the CLI command **show clock**, and then exit. Exiting a spawned CLI session returns me from whence I came—the

Bash shell:

```
[admin@Arista-7280 ~]$ Cli
Arista-7280>sho clock
Mon Nov 14 19:58:21 2016
Timezone: UTC
Clock source: local
Arista-7280>exit
[admin@Arista-7280 usr]$
```

The **Cli** command has some pretty interesting options. Just like most other Linux commands, I can see them by appending **--help** at the command line:

```
[admin@Arista-7280R ~]$ Cli --help
usage: Cli [-h] [-s SYSNAME] [-k SYSDBSOCKNAME] [-l] [--pdb] [-c COMMAND]
[-A]
      [-M] [-e] [-E] [-p PRIVILEGE] [-i PLUGINS] [-I] [-G]
      [--startup-config] [--disable-autocomplete]
      [-T STANDALONE_SHELL_TIMEOUT] [--completions COMPLETIONS]
      [config-filename]

[--output truncated--]
```

One of the more interesting options is the **-c command** or **--command=command** choice. Using these options, I can execute CLI commands from within Bash. For example, while in Bash, executing **Cli -c “sho ver”** spawns a CLI process, executes the CLI command **show version**, and then exits, reporting the output to **stdout**:

```
[admin@Arista-7280R ~]$ Cli -c "sho ver"
Arista DCS-7280SR-48C6-M-F
Hardware version: 21.05
Serial number: SSJ17290598
System MAC address: 2899.3abe.9f92

Software image version: 4.21.1F
Architecture: i386
Internal build version: 4.21.1F-9887494.4211F
```

```
Internal build ID: 1497e24b-a79b-48e7-a876-43061e109b92
Uptime: 0 weeks, 0 days, 4 hours and 32 minutes
Total memory: 32458980 kB
Free memory: 30390680 kB
```

Because this is Linux, I can pipe other commands, too. Here, I use `grep` to show only the line containing the word *image*:

```
[admin@Arista-7280R ~]$ Cli -c "sho ver" | grep image
Software image version: 4.21.1F
```

AMUSING DIGRESSION

In the first edition of *Arista Warrior*, I wrote that most of us who use these features don't use `Cli`, but rather the `FastCli` command. It behaves the same way, but instead of spawning a process (and taking upward of 10 seconds to do so), `FastCli` sends its requests to an agent called `FastClid` that processes our request and sends the output back. This is generally done in two seconds or less, which is why it's called `FastCli`.

In modern versions of EOS, `Cli` is actually a pointer to `FastCli`, so all `Cli` interactions are now fast. Also, thanks to years of advancement, it pretty much never takes two seconds anymore.

For my next trick, I redirect the output to a file:

```
[admin@Arista-7280R ~]$ Cli -c "show ver" | grep image > GAD.txt
[admin@Arista-7280R ~]$
```

I should now have a file in my home directory named *GAD.txt* that contains the output from my command. Let's take a look:

```
[admin@Arista-7280R ~]$ ls
GAD.txt
```

Sure enough, there it is. Using `cat` should work, and it does:

```
[admin@Arista-7280R ~]$ cat GAD.txt
Software image version: 4.21.1F
```

Be careful here, though! Writing files to my home directory is great, but I learned the hard way that anything written to the filesystem does not survive a reboot.

WARNING

That's worth a more prominent warning. Anything you write to the filesystem will not survive a reboot. There are only a few directory structures that remain untouched by a reboot: `/mnt/flash`, `/mnt/ usb1` (if installed), and the solid-state drive (SSD) drive (`/mnt/drive`), if your switch has one. There is also a `/persist/` directory in the root of the Bash filesystem, which is a link to the `/mnt/flash/persist/` directory. If you want the output of your scripts or commands to be saved after a reboot, you must store them in one of these locations. You have been warned!

Just as I could run a CLI command through the `Cli` command in Linux, I can run Bash commands from the `bash` command in CLI. Sure, that might sound like circular logic, but let me show you what I mean.

Remember how I got into Bash from CLI? I typed the command `bash`:

```
Arista-7280R#bash
Arista Networks EOS shell
[admin@Arista-7280R ~]$
```

That's pretty cool, but what if I just need the output of a single command and don't want to go through the hassle of dropping into Bash, executing the command, and exiting again? Good news! I can execute Bash commands from the CLI, without actually dropping to the Bash command line. All I need to do is append the Linux command

that I want to run.

Suppose that I want to get the output of the Linux command `uname -a`. To do this from the CLI, all I need to do is issue the command `bash uname -a`. This returns the output from the Unix command to me without ever leaving the CLI:

```
Arista-7280R#bash uname -a
Linux Arista-7280R 3.18.28.Ar-9854397.4211F #1 SMP PREEMPT Sun Sep 16
08:35:31
PDT 2018 x86_64 x86_64 x86_64 GNU/Linux
```

Note that any commands that you execute will be relative to your home directory. Thus, logged in as admin, if I ask for my current directory with the Unix `pwd` command, I will get the following results:

```
Arista-7280R#bash pwd
/home/admin
```

This book is loaded with examples in which I use Bash commands through the CLI, or use the Bash shell. After you get the hang of how this works, you'll begin to appreciate the power inherent in the design of Arista switches. When you feel the power, you'll cringe every time you need to use another vendor's switch. I know I do.

Some Quick EOS Bash Tips

The flash drive is located in `/mnt/flash`:

```
Arista(config)#dir
Directory of flash:/
-rwx 700978970 Nov 12 2018 EOS-4.21.1F.swi
-rwx 27 Feb 3 20:17 boot-config
```

```

drwx        4096          Feb  3 20:27  debug
drwx        4096          Feb  3 20:27  persist
drwx        4096          Feb  3 20:29  schedule
-rwx        1542          Feb  3 20:23  startup-config
-rwx         19           Feb  3 20:23  zerotouch-config

3440762880 bytes total (2034659328 bytes free)
Arista(config)#bash ls -l /mnt/flash
total 684576
-rwxrwx--- 1 root eosadmin 700978970 Nov 12 19:51 EOS-4.21.1F.swi
-rwxrwx--- 1 root eosadmin          27 Feb  3 20:17 boot-config
drwxrwx--- 3 root eosadmin        4096 Feb  3 20:27 debug
drwxrwx--- 2 root eosadmin        4096 Feb  3 20:27 persist
drwxrwx--- 3 root eosadmin        4096 Feb  3 20:29 schedule
-rwxrwx--- 1 root eosadmin        1542 Feb  3 20:23 startup-config
-rwxrwx--- 1 root eosadmin          19 Feb  3 20:23 zerotouch-config

```

If you have an SSD drive installed, it's called *drive*: in EOS and is located in */mnt/drive* in Bash:

```

Arista(config)#dir drive:
Directory of drive:/

-rw-        7168          Feb  3 20:25  aquota.user
drwx        4096          Feb  3 20:27  archive
drwx        16384         Dec  2  2016  lost+found
drwx        4096          Dec 17  2018  var_archive.2018-12-17-
23:15:02.dir
drwx        4096          Dec 18  2018  var_archive.2018-12-18-
18:01:01.dir
drwx        4096          Dec 18  2018  var_archive.2018-12-18-
18:11:04.dir
drwx        4096          Dec 18  2018  var_archive.2018-12-18-
18:15:02.dir
drwx        4096          Dec 18  2018  var_archive.2018-12-18-
00:01:02.dir
drwx        4096          Jan 11 14:05  var_archive.2019-01-01-
14:06:03.dir
drwx        4096          Jan 11 14:08  var_archive.2019-01-01-
14:15:02.dir
drwx        4096          Jan 13 09:48  var_archive.2019-01-13-
09:49:02.dir
drwx        4096          Jan 13 09:51  var_archive.2019-01-19-
10:01:02.dir
drwx        4096          Jan 20 15:47  var_archive.2019-01-20-

```

```
15:48:02.dir
    drwx        4096      Jan 20 15:51  var_archive.2019-02-01-
00:01:02.dir
    drwx        4096      Feb  3 20:22  var_archive.2019-02-03-
20:23:02.dir

125891358720 bytes total (125092859904 bytes free)
```

```
Arista(config)#bash ls -l /mnt/drive
total 76
-rw----- 1 root      root    7168 Feb  3 20:25 aquota.user
drwxr-xr-x 4 archive  archive 4096 Feb  3 20:27 archive
drwx----- 2 root      root   16384 Dec  2 2016 lost+found
drwxr-xr-x 4 archive  archive 4096 Dec 17 23:06 v[...]ive.2018-12-17-
23:15:02.dir
drwxr-xr-x 4 archive  archive 4096 Dec 18 17:59 v[...]ive.2018-12-18-
18:01:01.dir
drwxr-xr-x 4 archive  archive 4096 Dec 18 18:10 v[...]ive.2018-12-18-
18:11:04.dir
drwxr-xr-x 4 archive  archive 4096 Dec 18 18:13 v[...]ive.2018-12-18-
18:15:02.dir
drwxr-xr-x 4 archive  archive 4096 Dec 18 18:25 v[...]ive.2019-01-01-
00:01:02.dir
drwxr-xr-x 4 archive  archive 4096 Jan 11 14:05 v[...]ive.2019-01-11-
14:06:03.dir
drwxr-xr-x 4 archive  archive 4096 Jan 11 14:08 v[...]ive.2019-01-11-
14:15:02.dir
drwxr-xr-x 4 archive  archive 4096 Jan 13 09:48 v[...]ive.2019-01-13-
09:49:02.dir
drwxr-xr-x 4 archive  archive 4096 Jan 13 09:51 v[...]ive.2019-01-19-
10:01:02.dir
drwxr-xr-x 4 archive  archive 4096 Jan 20 15:47 v[...]ive.2019-01-20-
15:48:02.dir
drwxr-xr-x 4 archive  archive 4096 Jan 20 15:51 v[...]ive.2019-02-01-
00:01:02.dir
drwxr-xr-x 4 archive  archive 4096 Feb  3 20:22 v[...]ive.2019-02-03-
20:23:02.dir
```

To see the version of EOS installed from Bash (useful in scripts) look in the */etc/Eos-release* file:

```
[admin@Arista ~]$ more /etc/Eos-release
Arista Networks EOS 4.21.1F
```

To see a cool display of what processes called what other processes, use `ps tree`:

```
[admin@Arista ~]$ pstree
systemd--EosOomAdjust
    |-ProcMgr-master---ProcMgr-worker---Aaa---3*[{Aaa}]
    |           |
    |           | -ConfigAgent---bash---pstree
    |           |           `--4*
    |           |
    |           [{ConfigAgent}]
    |           |
    |           | -Fru
    |           | -IgmpSnooping
    |           | -Launcher
    |           | -PhyEthtool
    |           | -Rib
    |           | -Sand
    |           | -SlabMonitor
    |           | -StageMgr
    |           | -SuperServer---2*
    |
    [{SuperServer}]
    |
    | -agetty
    | -conlogd---sh---sed
    |           |
    |           `--tail
    |
    [--output removed--]
```

To see the interface names as they appear to the kernel in Bash, use the `ifconfig -s` command:

```
admin@Arista mnt]$ ifconfig -s
Iface      MTU RX-OK RX-ERR RX-DRP RX-OVR TX-OK TX-ERR TX-DRP TX-
OVR Flg
cpu        10209    0     0     0 0     0     0     0
0 BMRU
cpudebug  10209    0     0     0 0     0     0     0
0 BMRU
et1        10178    0     0     0 0     90860   0     0
0 BMRU
et2        10178    0     0     0 0     0     0     0
0 BMU
et3        10178    0     0     0 0     0     0     0
0 BMU
et4        10178    0     0     0 0     0     0     0
```

```

0 BMU
et5      10178      0      0      0 0      0      0      0
0 BMU
et6      10178      0      0      0 0      0      0      0
0 BMU
et7      10178      0      0      0 0      0      0      0
0 BMU

[ --output removed -- ]

mirror4 10209      0      0      0 0      0      0      0
0 BMRU
mirror5 10209      0      0      0 0      0      0      0
0 BMRU
mirror6 10209      0      0      0 0      0      0      0
0 BMRU
mirror7 10209      0      0      0 0      0      0      0
0 BMRU
mirror8 10209      0      0      0 0      0      0      0
0 BMRU
mirror9 10209      0      0      0 0      0      0      0
0 BMRU
mirror10 10209     0      0      0 0      0      0      0
0 BMRU
mirror11 10209     0      0      0 0      0      0      0
0 BMRU
mirror12 10209     0      0      0 0      0      0      0
0 BMRU
mirror13 10209     0      0      0 0      0      0      0
0 BMRU
mirror14 10209     0      0      0 0      0      0      0
0 BMRU
mirror15 10209     0      0      0 0      0      0      0
0 BMRU
txfwd    10209      0      0      0 0      0      0      0
0 BMRU
txraw    10209      0      0      0 0      0      0      0
0 BMRU
vxlan    10209      0      0      0 0      0      0      0
0 BMRU

```

Conclusion

Why use Bash? You absolutely don't need to in order to keep using the

switch the way we all have for decades, but if you really want to be an expert in the Arista ecosystem, you will find yourself returning to Bash time and time again. In fact, I recommend learning more about Linux to everyone who wants to learn more about Arista.

Chapter 13. Zero-Touch Provisioning

When a fixed-configuration Arista switch boots and no *startup-config* is found, the switch defaults to Zero-Touch Provisioning (ZTP) mode. Your first reaction to ZTP might be that it's a pain in the ass, but I assure you it's not, and I hope that by the end of this chapter, you'll agree. In fact, it's a seriously cool feature that you can use to great effect.

NOTE

Technically what triggers ZTP (when it's enabled) is a missing *startup-config* or one that is zero bytes in size.

Have you ever installed a new switch out of the box? Chances are that you mounted it and sat in the data center with a console cable, or you sat with it on your desk while your workmates plotted against you because of all the fan noise. Or consider the idea of remote installations. I've had many clients who have bought remote "smart" hands service, only to discover that those remote hands weren't so smart after all. ZTP is designed to provide the ability to eliminate both situations, all through the use of standards-based Dynamic Host Configuration Protocol (DHCP).

The reason I say that it can be a pain is because when a new switch is powered up and steps to use ZTP have not been taken, configuring the switch is next to impossible. Here's the first indication that you're in for a long day if you have no idea how Arista switches behave. When you press Enter at the login prompt of a new switch, you're greeted with the following warning:

```
localhost login:  
No startup-config was found.  
  
The device is in Zero Touch Provisioning mode and is attempting to  
download the startup-config from a remote system. The device will not  
be fully functional until either a valid startup-config is downloaded  
from a remote system or Zero Touch Provisioning is cancelled.  
  
To cancel Zero Touch Provisioning, login as admin and type  
'zerotouch cancel' at the CLI. Alternatively, to disable Zero Touch  
Provisioning permanently, type 'zerotouch disable' at the CLI.  
Note: The device will reload when these commands are issued.
```

Now, if you're a high-level know-it-all network guy like me, you glossed over that because there were so many words. Allow me to point out the important bits in *italic*:

The device will not be fully functional until either a valid startup-config is downloaded from a remote system or Zero Touch Provisioning is cancelled. Yeah, that looks important—especially the part about the device not being fully functional. Still didn't read? Allow me to demonstrate. Ignoring all dire warnings of impotent switches, I've logged in and started configuring. First, the hostname; it will need to be something special:

```
localhost#conf  
localhost(config)#hostname Arista
```

```
Arista(config)#
```

Cool! Now that my switch has a clever hostname, I save the configuration—because 30 years of bad habits are hard to undo:

```
Arista(config)#wri
```

NOTE

As of EOS version 4.21.1F, when you try to do this, EOS will throw an error stating, *Cannot copy to startup-config when ZeroTouch is enabled*, which is another example of the developers quickly working to make all of my writing obsolete. In this case, I must give them credit because I think this is a great change.

Life is good, so I take a well-deserved break for my hourly dose of caffeine and sugar, after which I come back and press Return.

Again, 30 years of bad habits force me to perform actions beyond my control, so without thinking I mash the Return key a couple of times:

```
Arista(config)#  
localhost(config)#  
localhost(config)#
```

What the hell? I configured the switch, even saved the configuration, and the switch reverted to a generic hostname on its own! Stupid new switches that...wait a minute. What was that error message I saw earlier? Wasn't there something about cancelling something?

Here's the part:

To cancel Zero Touch Provisioning, login as admin and type

'zerotouch cancel' at the CLI. Alternatively, to disable Zero Touch Provisioning permanently, type 'zerotouch disable' at the CLI.
Note: The device will reload when these commands are issued.

This little comedy of errors I just walked you through is almost a word-for-word account of my first interaction with an Arista switch. Because I'm a man, and a high-level networking know-it-all, I found no reason to read the documentation, let alone the warning message. Let's cut to the chase, and I'll show you what's really going on.

NOTE

Over years of teaching and helping networking engineers ranging from absolute newcomers to triple CCIEs, I've come to the conclusion that networking engineers are genetically predisposed to being incapable of reading warning message on screens.

When an Arista switch boots for the first time, ZTP is enabled. The switch sees that there is no *startup-config* so it configures all interfaces with the `no switchport` command and then sends out DHCP queries on any Ethernet and management interfaces that are connected. If you're patient (I'm not) and take the time to read the messages (I didn't), you'll see exactly what's going on. Here's a sample from a switch in such a state:

```
Apr 10 23:17:19 localhost ZeroTouch: %ZTP-5-DHCP_QUERY: Sending DHCP
request on [ Ethernet10, Ethernet11, Ethernet24 ]
Apr 10 23:18:19 localhost ZeroTouch: %ZTP-5-DHCP_QUERY_FAIL: Failed
to get a valid DHCP response
Apr 10 23:18:19 localhost ZeroTouch: %ZTP-5-RETRY: Retrying Zero Touch
Provisioning from the begining (attempt 2)
```

Different revs of code might have different output. On my 7280R

running EOS 4.21.1F, I see this output as the switch tries to get the interfaces to a default condition:

```
localhost(config)#Jan  7 20:15:34 localhost ConfigAgent: %ZTP-6-RETRY:  
Retrying  
  Zero Touch Provisioning from the beginning (attempt 1)  
Jan  7 20:15:34 localhost ConfigAgent: %ZTP-6-INTERFACE_SPEED: Setting  
interface  
  Speed to default speed for [Ethernet26, Management1, Ethernet27,  
Ethernet7,  
  Ethernet34, Ethernet8, Ethernet36, Ethernet46, Ethernet35, Ethernet22,  
Ethernet45, Ethernet12, Ethernet2, Ethernet42, Ethernet20, Ethernet18,  
Ethernet29, Ethernet17, Ethernet9, Ethernet41, Ethernet30, Ethernet40,  
Ethernet38, Ethernet16, Ethernet37, Ethernet4, Ethernet43, Ethernet1,  
Ethernet32, Ethernet23, Ethernet15, Ethernet3, Ethernet5, Ethernet24,  
Ethernet14, Ethernet11, Ethernet19, Ethernet39, Ethernet6, Ethernet48,  
Ethernet31, Ethernet47, Ethernet13, Ethernet10, Ethernet44, Ethernet21,  
Ethernet28, Ethernet25, Ethernet33]  
Jan  7 20:15:34 localhost ConfigAgent: %ZTP-6-INTERFACE_SPEED: Setting  
interface  
  speed to forced 10gfull for [Ethernet52/1, Ethernet49/3, Ethernet51/3,  
Ethernet54/3, Ethernet50/2, Ethernet51/1, Ethernet50/3, Ethernet49/1,  
Ethernet51/4, Ethernet52/3, Ethernet53/1, Ethernet53/4, Ethernet50/4,  
Ethernet50/1, Ethernet51/2, Ethernet52/2, Ethernet53/2, Ethernet49/4,  
Ethernet52/4, Ethernet54/4, Ethernet49/2, Ethernet54/2, Ethernet53/3,  
Ethernet54/1]
```

The switch found three connected interfaces (E10, E11, and E24) and sent DHCP queries out to all of them. Sadly, there were no responses, so it kept trying. If you leave it in this state, it will try forever. If you configure the switch, that configuration will be trashed as ZTP tries again and again to find a configuration, or at least an IP address.

At this point, there are four things you can do:

- Cancel ZTP
- Disable ZTP
- Actually boot using ZTP to your advantage

- Give up, quit your job, move to Texas, and grow sugar beets

I suppose we could also turn off the switch and hit the bar for some cocktails, but let's keep this discussion focused on ZTP and examine each choice. Well, maybe not the one about the sugar beets.

ZTP Requirements

For ZTP to work, you need the following:

- A DHCP server
- A server to host the configuration or script to be fetched (can be the same server)
- Connected interfaces on the switch

This feature works by sending a DHCP request out of every connected interface. When an IP address is supplied by DHCP, it's assigned to the interface on which the reply was received. The DHCP server must support the ability to provide a URL pointing to a file (which most modern DHCP servers do), which the switch will then retrieve. For now, let's assume that the file being retrieved is a configuration file. We talk about scripts a bit later in this chapter.

Cancelling ZTP

Cancelling ZTP is pretty simple. Just log in, enter configuration mode, and issue the `zerotouch cancel` command:

```
localhost(config)#conf
Jan  7 20:16:04 localhost ConfigAgent: %ZTP-6-RETRY: Retrying Zero Touch
Provisioning from the beginning (attempt 2)
```

```
localhost(config)#zerotouch cancel
Jan  7 20:16:08 localhost ConfigAgent: %ZTP-6-CANCEL: Cancelling Zero
Touch
  Provisioning
Jan  7 20:16:08 localhost Flushing AAA accounting queue: [  OK  ]

Restarting system [20:16:10]
```

Notice that there is no “Are you sure?” prompt. If you issue this command, the switch *will* reload. That’s not such a bad thing, because anything we tried to configure was wiped out anyway. Besides, those ZTP messages get old quickly.

When the switch reboots, you’ll be able to configure it as you would expect. Here’s the rub: if you cancel ZTP, it’s still there; it’s just not bothering you this time. In fact, the configuration will still have the login banner programmed with the ZTP warning message.

If, with ZTP cancelled, you erase the *startup-config* and reboot, you will again be treated to ZTP’s never-ending messages and attempts at finding a DHCP server:

```
May 14 05:44:59 localhost ZeroTouch: %ZTP-5-INIT: No startup-config
found, starting Zero Touch Provisioning
May 14 05:45:34 localhost ZeroTouch: %ZTP-5-DHCP_QUERY: Sending DHCP
request on [ Ethernet10, Ethernet11, Ethernet24 ]
```

The way I like to describe `zerotouch cancel` is that it disables ZTP from taking over for only one reboot.

Suppose that for some reason, no matter what, you don’t want to see ZTP’s ugly face again. For that to happen, you must completely disable ZTP.

Disabling ZTP

To completely disable ZTP, log in as admin and issue the `zerotouch disable` command. The switch will immediately reboot (again, without warning or confirmation), only this time, ZTP is terminated with extreme prejudice as evidenced by the following messages:

```
localhost#zerotouch disable
May 14 05:47:42 localhost ZeroTouch: %ZTP-5-CANCEL: Cancelling Zero Touch
Provisioning
May 14 05:47:42 localhost ZeroTouch: %ZTP-5-RELOAD: Rebooting the system
May 14 05:47:43 localhost ProcMgr-worker: %PROCMGR-6-WORKER_WARMSTART:
ProcMgr worker warm start. (PID=1446)
May 14 05:47:43 localhost ProcMgr-worker:
%PROCMGR-6-TERMINATE_RUNNING_PROCESS: Terminating
deconfigured/reconfigured process 'ZeroTouch' (PID=1506)
May 14 05:47:44 localhost ProcMgr-worker:
%PROCMGR-6-PROCESS_TERMINATED: 'ZeroTouch' (PID=1506) has terminated.
May 14 05:47:44 localhost ProcMgr-worker:
%PROCMGR-6-PROCESS_NOTRESTARTING: Letting 'ZeroTouch' (PID=1506)
exit - NOT restarting it.
```

Now, the system boots once more, and there isn't even a trace of ZTP:

```
Aboot 6.1.2-4757975

Press Control-C now to enter Aboot shell
Booting flash:/EOS-4.21.1F.swi
[    8.431535] Starting new kernel
[   1.651605] Running dosfsck on: /mnt/flash
Switching rootfs

Welcome to Arista Networks EOS 4.21.1F
New seat seat0.
RTNETLINK answers: No such process
[   71.304306] EXT4-fs (sda): VFS: Can't find ext4 filesystem
[   71.379014] EXT4-fs (sda): VFS: Can't find ext4 filesystem
[   71.457489] FAT-fs (sda): invalid media value (0xf3)
Arista File Archive: initialization complete, quotapct: 20
warning: file /usr/share/doc/strongswan-5.3.0/NEWS: remove failed: No
such
```

```
file or directory
[ OK ] ConnMgr: Starting TimeAgent: [ OK ]
[ OK ]
Starting ProcMgr: [ OK ]
Starting EOS initialization stage 1: [ OK ]
Starting NorCal initialization: [ OK ]
Starting EOS initialization stage 2: [ OK ]
Starting Power OCompleting EOS initialization (press ESC to skip): [ OK ]
]
Model: DCS-7280SR-48C6-M
Serial Number: SSJ17290598
System RAM: 32458980 kB
Flash Memory size: 3.1G

localhost login:
```

At this point you can reboot the switch to your heart's content, and ZTP will not come back. If you'd like to bring ZTP back, you can't even do it from `exec` mode:

```
localhost#zerotouch ?
cancel          Cancel ZeroTouch and reload the switch
disable         Disable ZeroTouch and reload the switch
script-exec-timeout  Change timeout for the downloaded script to
                     finish execution
```

In fact, you can't even do it in `config` mode, though there's an option for `enable`, which gives us a clue as to what we need to do:

```
localhost#conf
localhost(config)#zerotouch enable
% Configuration ignored: ZeroTouch can not be enabled interactively.
To enable ZeroTouch, delete startup-config and reload the switch.
```

Sadly, that message is not entirely correct—at least up to EOS version 4.21.1F. Unless you accept the fact that nothing is impossible with an Arista switch! Check out what's on the `flash:` drive:

```
localhost#dir
Directory of flash:/
```

```
-rwx 700978970 Nov 12 2018 EOS-4.21.1F.swi
-rwx 701781780 Nov 12 2018 EOS-4.21.2F.swi
-rwx 77 Jan 7 20:44 SsuRestore.log
-rwx 27 Jan 4 17:04 boot-config
drwx 4096 Jan 7 20:46 debug
drwx 4096 Jan 7 20:46 persist
drwx 4096 Jan 2 22:09 schedule
-rwx 0 Jan 7 20:33 startup-config
-rwx 13 Jan 7 20:42 zerotouch-config

3269361664 bytes total (1157074944 bytes free)
```

Hmm. A file named *zerotouch-config* exists. Let's dig into that file and see what we find, shall we?

```
localhost#more flash:zerotouch-config
DISABLE=True
```

Ha! I'll bet if we change that to false, we'll get ZTP back. In fact, in the first edition of *Arista Warrior*, I did just that, but years of maturity have set in, and I now recommend that people just delete the file, which has the same effect and doesn't get your hands dirty with vi:

```
localhost#del flash:zerotouch-config
localhost#
```

Because the *startup-config* is zero bytes, I don't need to `write erase`, so I'll force a reboot to see what happens:

```
localhost(config)#reload now

Broadcast message from root@localhost (Mon Jan 7 20:51:37 2019):
The system is going down for reboot NOW!
Flushing AAA accounting queue: [ OK ]

Restarting system
[--- lots of output truncated ---]
May 14 19:42:54 localhost ZeroTouch: %ZTP-5-INIT: No startup-
```

```
config found,  
starting Zero Touch Provisioning
```

This was actually pretty cool for me when I first figured it out (back around 2012 or so) because I could find no mention of how to do this in the documentation, and with my experience using Arista switches I reasoned that there had to be a configuration file somewhere and that Arista would never permanently disable a feature. Following that logic, I found the configuration file, changed it, and was rewarded with the expected results. Today I have access to all sorts of nifty Arista documentation both internal and external, but I can't show you any of that secret stuff.

I feel that it's important to point out that ZTP is not controlled in any way by the *startup-config*, and if you think about it, it cannot be. Why? Because the feature is triggered by the lack of the file, so how could it be configured using a file that needs to not be there for it to work? That's why ZTP is configured with a file on *flash*:

If you're wondering how `zerotouch cancel` works, when the switch reboots, it puts the string `disable=once` or `disable=nextReload` into the *boot-config*, depending on the version of EOS. This lets the switch come up normally, and when it does, one of the first things it does is delete that file so that the next time it boots, ZTP is triggered once more assuming no *startup-config*.

Booting with ZTP

Hopefully you've considered the possibility of booting with ZTP, because as we're about to see, it's pretty darn cool. As with all things

Arista, it's well thought out and powerful.

The first thing we need to know is that while ZTP can load a configuration from the network, it can also run scripts. Before we get to that, let's take a look at what needs to happen in order for an Arista switch to boot using ZTP.

When a switch boots in ZTP mode, it configures all of the Ethernet and Management interfaces with the `no switchport` command in order to allow DHCP to be run on those interfaces. DHCP is capable of much more than just providing an IP address, and ZTP takes advantage of this fact. To see how, let's first take a look at how to configure a DHCP server to use ZTP.

I've set up a lab in which I have an Ubuntu Linux server, an existing network, and a new Arista switch. I'm using `dhcpd` on the Linux server to serve DHCP. The examples used in this section reflect the configuration required for `dhcpd`, though the concepts should be universal for any DHCP server that supports them.

To begin, I configure the regular DHCP parameters as follows to fit with my network:

```
subnet 10.0.0.0 netmask 255.255.255.0 {
    option subnet-mask 255.255.255.0;
    option broadcast-address 10.0.0.255;
    option domain-name-servers 10.0.0.100;
    option domain-name "gad.net";
    option bootfile-name "http://10.0.0.100/ZTP/ZTP-Script";
    pool {
        range 10.0.0.30 10.0.0.50;
    }
}
```

Given this configuration, dynamically allocated IP addresses will be served from the pool 10.0.0.30–50. This configuration will deliver an IP address from the pool to any device that requests an IP address. Lastly, the `option bootfile-name` line shows an HTTP URL. The switch, upon receiving and activating the IP address, will go to this URL and download the bootfile. Here's where this gets fun.

This file can be either a *startup-config*, in which case the switch will apply the configuration and reboot, or it can be a script. ZTP will decide based on the first line of the file found at the `bootfile-name` address. If the first line includes a *shebang* (`#!/`) and the path to an interpreter, that interpreter will be loaded and the file will be considered a script. If not, the file will be considered a *startup-config* and treated accordingly.

Let's look at an example that might apply for the real world. Here's how I've configured my *ZTP-Script* file on the web server:

```
[root@cozy ZTP]$ more ZTP-Script
#!/usr/bin/CLI
enable
copy http://www2.gad.net/Arista/Arista1-startup flash:startup-config
copy http://www2.gad.net/Arista/EOS-4.21.1F.swi flash:
config
boot system flash:EOS-4.21.1F.swi
```

This file will perform the following actions on the switch, in order, after ZTP loads it:

- Load the CLI process
- Enter EOS CLI enable mode
- Copy the *startup-config* from the web server to *flash*:

- Copy the desired revision of EOS from the web server to *flash*:
- Enter EOS CLI configuration mode
- Set the *boot-config* to boot from the new EOS version just downloaded

NOTE

The script does not have a `wr mem` or `copy run start` at the end of it. Does it need one? Is that a typo? Do I really have no idea what I'm doing as has been postulated by those who think I'm wrong? If you think that there needs to be a `wr` at the end of that script, I'd advise two things: 1) read the script again, and 2) read the next chapter, which happens to be about Aboot.

Would a `wr` at the end of the script hurt anything? Nope! But I like to get people to think about how things work.

After the script has completed, the switch will reload. Let's take a look at the switch as it boots with the ZTP configuration loaded on the DHCP server.

Here, my switch is running EOS version 4.19.1F and has no configuration:

```
localhost#sho ver
Arista DCS-7280SE-72-F
Hardware version: 01.00
Serial number: JPE15181350
System MAC address: 001c.7390.93cf
Software image version: 4.19.1F
Architecture: i386
Internal build version: 4.19.1F-3205780.4166M
Internal build ID: 373dbd3c-60a7-4736-8d9e-bf5e7d207689
```

Uptime:	3 minutes
Total memory:	3844356 kB
Free memory:	470840 kB

And the existing boot configuration is pointing to 4.19.1F, as well:

```
localhost#sho boot-config
Software image: flash:/EOS-4.19.1F.swi
Console speed: (not set)
Aboot password (encrypted): (not set)
Memory test iterations: (not set)
```

Now I erase the *startup-config* with the `write erase` command:

```
localhost#write erase
Proceed with erasing startup configuration? [confirm]
localhost#
```

And reload with extreme prejudice. Or something:

```
Arista#reload now

Broadcast message from root@localhost
          (unknown) at 5:13 ...

The system is going down for reboot NOW!
```

Now let's watch the ZTP messages as the switch boots (I've included only the relevant message here to save space):

```
Address: 10.0.0.45/24/24; Nameserver: 10.0.0.100; Domain: gad.net;
Boot File: http://10.0.0.100/ZTP/ZTP-Script ]
Jan  7 21:05:23 localhost ConfigAgent: %ZTP-6-CONFIG_DOWNLOAD: Attempting
to
download the startup-config from http://10.0.0.100/ZTP/ZTP-Script
Jan  7 21:05:23 localhost ConfigAgent: %ZTP-6-CONFIG_DOWNLOAD_SUCCESS:
Successfully downloaded config script from http://10.0.0.100/ZTP/ZTP-
Script
Jan  7 21:05:23 localhost ConfigAgent: %ZTP-6-EXEC_SCRIPT: Executing the
downloaded config script
Jan  7 21:05:24 localhost ZTP: : Interface found:  1
```

```
Jan  7 21:05:24 localhost ConfigAgent: %ZTP-6-EXEC_SCRIPT_SUCCESS:  
Successfully  
executed the downloaded config script  
Jan  7 21:05:24 localhost ConfigAgent: %ZTP-6-RELOAD: Rebooting the  
system  
Flushing AAA accounting queue: [  OK  ]
```

At this point the switch has downloaded the ZTP file, applied what it found there, and reloaded. So, without direct interaction, the switch booted, ran our script, loaded the new configuration, downloaded new code, rebooted, and applied the new code.

```
ZTP-Switch#sho ver  
Arista DCS-7280SR-48C6-M-F  
Hardware version: 21.05  
Serial number: SSJ17290598  
System MAC address: 2899.3abe.9f92  
  
Software image version: 4.21.1F  
Architecture: i386  
Internal build version: 4.21.1F-9887494.4211F  
Internal build ID: 1497e24b-a79b-48e7-a876-43061e109b92  
  
Uptime: 0 weeks, 0 days, 0 hours and 10 minutes  
Total memory: 32458980 kB  
Free memory: 30521916 kB
```

The hostname `ZTP-switch` came from the copied `startup-config`, and the switch is now running on a new revision of code.

Imagine a scenario in which you're installing 50 new Arista switches. This procedure could put a base configuration on them all while also loading the right version of code. That's my kind of time-saving feature! Oh, and if you're using CloudVision, this is *exactly* how that tool configures new switches! CloudVision has a ZTP server built in that handles all of this for you. Note that you need to enable the DHCP server on CloudVision if you want to use it, and this is because you can

also use your own DHCP server if, for example, you already have one in place.

If you don't like writing your own code and you don't have CloudVision, there's another option (Arista loves options!) called *ztpserver* that you can get from the EOS Plus team at its [GitHub page](#).

Not only that, but some customers have developed scripts that autoarchive the configurations on all of their switches every day (which is always a good idea) and then update the ZTP files accordingly. When a switch fails, ZTP on the replacement switch references the proper files that have been scripted to contain the configuration of the switch being replaced. Think about the time saved during an outage when the switch configures itself after a replacement.

Conclusion

In my training labs at Arista, I've written a ZTP script that allows the switches to configure themselves based on their position within the network. How? By issuing a `show lldp neighbor` ([Chapter 11](#)) command that allows the switch to know what interface it's connected to on the management switch. If the switch is connected to interface e5 on the management switch, it configured itself as student-05. You can see this script on my [GitHub page](#). The ability to have the switch run a script in order to configure itself is a fabulous capability, but remember that if you don't like to code, you can use ZTP server or CloudVision ([Chapter 15](#)) to accomplish something similar.

Chapter 14. Aboot

Aboot is the bootloader for EOS. The bootloader is a small program that loads automatically when the system is powered on. The bootloader's primary job is to load the primary operating system (OS), which is usually stored elsewhere, such as flash memory or disk. If you've ever installed a Linux system, you've likely encountered bootloaders such as GRUB. On Windows NT/2000/XP machines, the default bootloader is NTLDR.

NOTE

Technically, there are two types of bootloaders: first stage and second stage. The first-stage bootloader usually operates at a very low level and is often responsible for hardware system checks. A PC's BIOS would be considered a first-stage bootloader, whereas the bootloaders mentioned in this chapter would be second-stage bootloaders.

Before EOS is loaded, the switch loads the Aboot process from ROM. Aboot looks for a file called *boot-config* in *flash:/*, which to be painfully accurate, is actually */mnt/flash* in Linux:

```
[admin@Arista ~]$ cd /mnt/flash
[admin@Arista flash]$ ls
EOS-4.20.1F.swi  debug      schedule      zerotouch-config
boot-config       persist    startup-config
```

Aboot reads the contents of this file, determines the image to be loaded, and then loads it. If the *boot-config* does not exist, Aboot will

halt the system and present an `Aboot#` prompt.

In fact, if any of the following should occur, Aboot will halt the system:

- `boot-config` is corrupt or not found
- Configured EOS image is corrupt or not found
- Ctrl-C is entered from the console while the Aboot process is running

Normally, the system boots as follows (details might differ depending on loaded modules and the version of EOS installed):

```
Aboot 6.1.2-4757975

Press Control-C now to enter Aboot shell
Booting flash:/EOS-4.20.1F.swi
[    8.336429] Starting new kernel
[   1.674818] Running dosfsck on: /mnt/flash
Switching rootfs

Welcome to Arista Networks EOS 4.20.1F
New seat seat0.
RTNETLINK answers: No such process
[   63.406686] EXT4-fs (sda): VFS: Can't find ext4 filesystem
[   63.483960] FAT-fs (sda): invalid media value (0xf3)
Arista File Archive: initialization complete, quotapct: 20
[ OK ] TimeAgent: Starting ConnMgr: [ OK ]
[ OK ]
Starting ProcMgr: [ OK ]
Starting EOS initialization stage 1: [ OK ]
Starting NorCal initialization: [ OK ]
Starting EOS initialization stage 2: [ OK ]
Starting Power OCompleting EOS initialization (press ESC to skip): [ OK ]
]
Model: DCS-7280SR-48C6-M
Serial Number: SSJ17290599
System RAM: 32459704 kB
Flash Memory size: 3.1G
```

```
Arista login:
```

By pressing Ctrl-C on the console when prompted, we interrupt the boot process and drop into the Aboot shell:

```
Aboot 6.1.2-4757975
Press Control-C now to enter Aboot shell
^CWelcome to Aboot.
Aboot#
```

While in Aboot, the fans in the switch run at high speed. They put out some significant noise in this state, so if you're playing with a switch on your desk at work, prepare for all of your local cube dwellers to hate you. If you work in an open office environment, I recommend leaving the switch in the office and connecting to it from home with a console server so that the noise won't bother you. That'll teach 'em.

Aboot has a **help** command that shows the following:

```
Aboot# help

Commonly-used Aboot commands

ls          Prints a list of the files in the current working directory
cd          Changes the current working directory
cp          Copies a file
more        Prints the contents of a file one page at a time
vi          Edits a text file
boot        Boots a SWI
swiinfo     Prints information about a SWI
recover     Recovers the factory-default configuration
reboot      Reboots the switch
netconf     Configures a network interface manually (IPv4 or IPv6)
udhcpc      Configures a network interface automatically via DHCP (IPv4
only)
wget        Transfers a file from an HTTP or FTP server
scp         Transfers a file to or from a server running SSH
showtech    Show system information
```

```
Run 'command -h' for brief help on a specific command.
See http://busybox.net/ for additional help on many commands.
```

Navigating around is easy if you're familiar with Linux (and you really should be). Although Aboot is Linux, it's a flavor of Linux called Busybox, which is an open source version of Unix utilities compiled into a single small executable designed to run in a USB stick. The thing to remember about Aboot is that it's a tiny Linux; if you think of it as such, you'll do fine. Unless you don't know anything about Linux, in which case you're screwed. Good thing you bought this book!

From within the Aboot prompt, the first thing we do is to try and get our bearings. When I'm lost on a Linux box, I issue the `pwd` command to see what directory I'm in. Sure enough, this works just fine in Aboot:

```
Aboot# pwd
/
```

So, we're in the root, which means I'm bored. Let's take a look around by using the `ls` command:

```
Aboot# ls
MD5SUMS  dev  init  mnt  root  tmp
bin      etc  lib   proc  sys
```

Looks harmless enough. I wonder if more elaborate versions of these commands work?

```
Aboot# ls -al
drwxr-xr-x  11 root  0          0 Apr 24 19:51 .
drwxr-xr-x  11 root  0          0 Apr 24 19:51 ..
-rw-r--r--   1 root  0          2012 Apr 19 2017 MD5SUMS
drwxr-xr-x   2 root  0          0 Apr 19 2017 bin
drwxr-xr-x   2 root  0          0 Apr 24 19:51 dev
```

drwxr-xr-x	2	root	0	0 Apr 24 19:51	etc
-rwxr-xr-x	122	root	0	448836 Apr 19 2017	init
drwxr-xr-x	2	root	0	0 Apr 19 2017	lib
drwxr-xr-x	4	root	0	0 Apr 24 19:51	mnt
dr-xr-xr-x	66	root	0	0 Apr 24 19:51	proc
drwx-----	2	root	0	0 Mar 22 2017	root
dr-xr-xr-x	11	root	0	0 Apr 24 19:51	sys
drwxrwxrwt	2	root	0	0 Apr 24 19:51	tmp

Yay! But this looks like any Unix machine. Where's the good stuff? Because I'm in Aboot, I'd probably want to check, change, or otherwise mangle the *boot-config*, and I know that resides in *flash:/* from within EOS, but that doesn't seem to exist here. That's because *flash:/* is an EOS construct. The key to mounted filesystems in Fedora Core Linux (the Linux used to build EOS) is */mnt*, so let's take a look there:

```
Aboot# cd mnt/
Aboot# ls -al
drwxr-xr-x 4 root 0 0 Apr 24 19:51 .
drwxr-xr-x 11 root 0 0 Apr 24 19:51 ..
drwxrwx--- 16 root 88 4096 Apr 24 19:48 drive
-rw-rw-rw- 1 root 0 91 Apr 24 19:51 drive.conf
drwxrwx--- 7 root 88 4096 Jan 1 1970 flash
-rw-r--r-- 1 root 0 94 Apr 24 19:51 flash-
original.conf
-rw-rw-rw- 1 root 0 90 Apr 24 19:51 flash-
recover.conf
-rw-rw-rw- 1 root 0 94 Apr 24 19:51 flash.conf
```

NOTE

I tend to repeat this bit about *flash:/* being an EOS construct because I've found that people not familiar with Unix find this a bit confusing. That, and I really like the word *construct*. It makes me feel like I'm in *Star Trek* when I say it out loud. Try it for yourself and see. *Construct...*

Looks promising! There's a directory within `/mnt` named *flash*, so let's see what's in there:

```
Aboot# cd flash/
Aboot# pwd
/mnt/flash
Aboot# ls -l
-rwxrwx--- 1 root 88 638234211 Nov 21 21:13 EOS-4.20.1F.swi
-rwxrwx--- 1 root 88 27 Apr 23 11:28 boot-config
drwxrwx--- 3 root 88 4096 Apr 24 19:51 debug
drwxrwx--- 2 root 88 4096 Apr 24 19:50 persist
drwxrwx--- 3 root 88 4096 Apr 23 11:40 schedule
-rwxrwx--- 1 root 88 2885 Apr 24 19:46 startup-config
-rwxrwx--- 1 root 88 19 Apr 24 15:14 zerotouch-config
```

Ah-ha! We've found the *flash*:/ location from within Aboot. In the future, we can just issue the `cd /mnt/flash/` command from within Aboot to get back here.

NOTE

The file structure in Aboot is pretty much the same as it would be in Bash, though any temporary file structures created when EOS boots will be missing. Generally, if you're in Aboot it's because something is wrong and to fix it, you'll likely need to `cd` to `/mnt/flash/`. Remember that your home directory does not survive a reboot, so if you're in Aboot looking for your home directory, you're out of luck.

To see the contents of a file in Linux, we might use the `more` command. Let's do exactly that in order to see what's contained within the *boot-config*:

```
Aboot# more boot-config
SWI=flash:/EOS-4.20.1F.swi
```

Not very exciting, is it? The single line indicates that the SWI file can be found at *flash:/EOS-4.20.1F.swi*. Note that this is configured using EOS reference points (*flash:/*), not Linux reference points (*/mnt/flash/*). My guess is that it's done this way to make it more palatable to network engineers unfamiliar with Linux. Now let's see what other sorts of trouble we can get ourselves into with this file. There are some cool options that we can configure in the *boot-config*:

SWI

Set the location of the SWI

CONSOLESPEED

Set the speed of the console port

PASSWORD

The encrypted password for the Aboot shell

NET commands

Set various configurations pertaining to simple network connectivity

These commands are placed within the *boot-config*, with the syntax **COMMAND= configuration**. This file can be examined from within Aboot or the Bash shell using the `more /mnt/flash/boot-config` command or from within EOS with the `show boot` CLI command:

```
Arista#sho boot
Software image: flash:/EOS-4.20.1F.swi
Console speed: (not set)
Aboot password (encrypted): (not set)
Memory test iterations: (not set)
```

We've already seen the **SWI** command in action, but let me just point

out that although the obvious method is to point to an image on flash, we can also point to images outside the box. Here are some cool examples of valid SWI statements:

SWI on flash

```
SWI=flash:EOS-4.20.1F.swi
```

If you notice, there is no slash after the colon in this example (*flash:* versus *flash:/*). Either will work.

SWI on USB1

```
SWI=usb1:/EOS-4.20.1F.swi
```

SWI on */mnt/flash* (same as *flash:/*)

Yes, you can use either *flash:* or */mnt/flash/*! Because this is usually configured with the CLI boot system command, this line will likely contain *flash:*

```
SWI=/mnt/flash/EOS-4.20.1F.swi
```

SWI on an HTTP server

```
SWI=http://foo.com/images/EOS-4.20.1F.swi
```

SWI on an FTP server

```
SWI=ftp://foo.com/EOS-4.20.1F.swi
```

SWI on an FTP server with a username and password

```
SWI=ftp://user:pass@foo.com/EOS-4.20.1F.swi
```

SWI on a TFTP server

```
SWI=tftp://foo.com/EOS-4.20.1F.swi
```

SWI on an NFS-mounted filesystem

```
SWI=nfs://foo.com/images/EOS-4.20.1F.swi
```

Although you can have your image reside somewhere on the network, I'm not a fan of this because for that to work, you'll need to configure IP information in the *boot-config*, as well. Not only that, but since we're in Aboot and not EOS, that means that the ASIC driver is not loaded, which means that the only available interface, is the physical management interface as shown by issuing the **ifconfig -a** command in Aboot:

```
Aboot# ifconfig -a
lo      Link encap:Local Loopback
        LOOPBACK MTU:16436 Metric:1
        RX packets:0 errors:0 dropped:0 overruns:0 frame:0
        TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:0
        RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

ma1     Link encap:Ethernet  HWaddr 28:99:3A:BE:9F:91
        BROADCAST MULTICAST  MTU:1500  Metric:1
        RX packets:0 errors:0 dropped:0 overruns:0 frame:0
        TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1000
        RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
        Interrupt:36
```

OK, you get the point. Now let's move on to the other *boot-config* commands.

The **CONSOLESPED** command is pretty simple. We can apply one choice from a list of speeds, and that's it. We cannot set stop bits, parity bits, or anything other than speed with this command. Values include common speeds for serial ports, including 1200, 2400, 4800, 9600, 19200, and 38400 with the default being 9600. Here is an example *boot-config* with the **CONSOLESPED** set to 38400:

```
Aboot# more boot-config
SWI=flash:/EOS-4.20.1F.swi
```

```
CONSOLESPEED=38400
```

You also can configure the CONSOLESPEED setting by using the EOS **boot console speed *speed*** command:

```
Arista(config)#boot console speed ?  
baud Console port speed (1200, 2400, 4800, 9600, 19200 or 38400)
```

The PASSWORD command is also pretty simple, but you should not configure it from within Aboot. You should set the PASSWORD command using only the EOS **boot secret** command because the EOS command encrypts the password, and the *boot-config* contains the encrypted result. With a password in place, you'll need to authenticate in order to access the Aboot shell, and this password cannot be recovered from Aboot, so assign an Aboot password with care.

Let's assign a password to Aboot from within EOS with the **boot secret** command. Here, I set the password to *Arista!*:

```
Arista(config)#boot secret Arista!
```

NOTE

I know I keep going back and forth between EOS and Aboot, and that might be confusing. You cannot go back and forth between these modes; I'm just showing how these items would be configured when within each of the modes. Stay with me, and watch the prompts carefully if you get mixed up as to where I am.

Now, viewing the *boot-config* with the **show boot** command, we can see that the file contains a PASSWORD entry and that the password is encrypted:

```
Arista(config)#sho boot
Software image: /mnt/flash/EOS-4.20.1F.swi
Console speed: (not set)Aboot password (encrypted):
$6$Mo6m6VkhGeG7.6pW$xFy0SEi2
EalgVy3xTjZagdGUMsEqxT31Mg.2YormEDdTtXCa0K2aPP1KwC/D0LUvpv0xx
HVbT77J6F6x8Jzik.
Memory test iterations: (not set)
```

In the first edition of this book, I wrote, “Whether or not the password is easily cracked is not germane to the subject matter of this book, and shame on you for thinking such things!” because of the MD5 encryption in use at the time, but as of about 2015 or so, EOS now defaults to SHA512, which is much more cryptographically sound. It also results in much longer encrypted password strings.

Upon a reboot, with the `PASSWORD` command set in the `boot-config`, we are now prompted for authentication when we press `Ctrl-C` at the `Aboot` message:

```
Restarting system
[00:48:22] watchdog punch .
[00:48:26] watchdog punch .
no FE found in the system
```

Aboot 4.0.4-2086886

Press Control-C now to enter Aboot shell^C

Aboot password:

At this point, if we enter the incorrect password three times, we are greeted with this friendly message:

```
Press Control-C now to enter Aboot shell^C
Aboot password:
incorrect password
```

```
Aboot password:  
incorrect password  
Aboot password:  
incorrect password  
Type "fullrecover" and press Enter to revert /mnt/flash to  
factory default state, or just press Enter to reboot:
```

If we feel the need to issue the `fullrecover` command, we are warned once again:

```
Type "fullrecover" and press Enter to revert /mnt/flash to factory  
Default state, or just press Enter to reboot: fullrecover  
All data on /mnt/flash will be erased; type "yes" and press Enter to  
proceed,  
or just press Enter to cancel:
```

Although it might not be obvious that “All data on `/mnt/flash` will be erased” is a bad thing, consider that the following files exist in `/mnt/flash`:

- The SWI files (you know, EOS and stuff)
- The *boot-config*
- The *startup-config*
- The *zerotouch-config*
- Some console logs on EOS 4.21 and later
- All schedule logs (unless stored elsewhere)
- Anything you might have put there
- Anything in the `/mnt/flash/persist/` folder

In other words, if you perform a `fullrecover` on the switch, your switch will lose everything, and you’ll need to start from scratch. Fun! Still, my pain is your gain, so let’s go ahead and see what happens:

```
Type "fullrecover" and press Enter to revert /mnt/flash to factory
default
state, or just press Enter to reboot: fullrecover
All data on /mnt/flash will be erased; type "yes" and press Enter to
proceed,
or just press Enter to cancel: yes
Erasing /mnt/flash
Writing recovery data to /mnt/flash
EOS-4.19.1F.swi
startup-config
boot-config
850926 blocks
[ 85.460223] Restarting system.
```

Ouch! We went from EOS version 4.20.1F to 4.19.1F, which might not seem like that big of a deal, but this is a pretty modern switch as of 2019. In the first edition of Arista Warrior we went from EOS version 4.8.1 to version 4.4.0 (the version my very old switch originally shipped with). Not only that, but our configuration is gone, the multiple EOS versions are gone from flash, and the switch is now a big unconfigured time sink. Luckily for you, that's my time being sunk, so while you sit back enjoying a cocktail, I'll be here rebuilding the switch.

NOTE

So, what's the moral of the story? Don't issue the `fullrecover` command unless you really mean it, because it doesn't just delete the `boot-config`. It deletes everything!

Why do it? `fullrecover` is the means to return your switch to factory default in the event of a lost Aboot password and lost EOS passwords without having to send the switch back to Arista.

Note that depending on your switch and depending on what version of

EOS you were originally on, you might get messages like this while booting a new version, regardless of if the change is an upgrade or a downgrade:

```
Starting NorCal initialization:  
-----  
Upgrading the casini system fpga.  
This process can take 6 minutes.  
Please do not reboot your switch.  
-----  
Upgrade of the casini system fpga completed successfully.  
Power cycling the system after successfully upgrading all  
system fpgas.
```

This is EOS putting new microcode onto the Field Programmable Gate Array (FPGA), which is essentially a programmable Application-Specific Integrated Circuit (ASIC). Some switches, like the 7280SE-72 shown in this example, contain multiple FPGAs and, as such, might take a fair bit of time to finish this task. Additionally, there might be multiple reboots while the system upgrades/downgrades, so this can take upward of 10 to 12 minutes for a single 1-rack unit (RU) switch.

Because I teach the Arista ACE classes, I get to listen to students as they watch their switches go through this process repeatedly. It takes a while, and no one likes to watch a switch boot, so do yourself a favor and grab another cocktail while it's going. Unless you're in my class, in which case, only the instructor can have cocktails—for safety reasons.

On a switch with a boot password that you want to remove, use the `no boot secret` command from within EOS. You could also just remove the `PASSWORD=` line in the *boot-secret* file from Bash:

```
Arista(config)#no boot secret
```

With nothing but the SWI set in the *boot-config*, you should see something like the following output when using the **show boot** command from within EOS:

```
Arista#sho boot
Software image: flash:/EOS-4.21.1F.swi
Console speed: (not set)
Aboot password (encrypted): (not set)
Memory test iterations: (not set)
```

Moving on, let's take a look at some of the NET commands within the Aboot environment. NET commands include the following:

NETDEV=interface

The interface that the switch will be configured to use for loading configurations or SWI files. This interface can only be an out-of-band management port, not one of the normal Ethernet switch interfaces. This is because the front-panel interfaces are controlled by the ASIC, and the ASIC driver has not been loaded in Aboot. On a fixed-configuration switch, the only choice is *ma1*.

NETAUTO=auto_setting

If using DHCP, this would be set to **dhcp**.

NETIP=interface_address

The IP address for the NETDEV interface.

NETMASK=interface_mask

The IP subnet mask for the NETDEV interface.

NETGW=gateway_address

The IP gateway address to allow the NETDEV interface to

communicate outside its directly connected IP network.

NETDOMAIN=domain_name

The DNS domain name for the switch.

NETDNS=dns_address

The IP address of a DNS server that can be used to resolve external hostnames.

Note that there is no support for IPv6 and that only dotted-decimal notation can be used (no /24 masks, for example). Also, be advised that these commands can be set only from within Aboot or from the Bash shell. They cannot be configured from within EOS:

```
Arista(config)#boot ?
  console  Console port settings
  secret   Assign the Aboot password
  system   Software image URL
  test     Boot test
```

Here's an example of how the *boot-config* might be configured on a simple network:

```
Aboot# more /mnt/flash/boot-config
SWI=tftp://10.0.0.100/OS/OS-4.20.1F.swi
NETDEV=ma1
NETIP=10.0.0.44
NETMASK=255.255.255.0
NETGW=10.0.0.1
NETDOMAIN=arista.com
NETDNS=10.0.0.100
```

When I first started messing with this file, I naturally placed all new commands where they belonged and then rebooted and dutifully pressed Ctrl-C as soon as the message commanded me to. Only, it didn't work! After about 90 tries, I wasn't paying attention and pressed

Ctrl-C later in the boot process, and that is when my Aboot network configuration worked. Here, I'll show you. First, let's reboot the switch and press Ctrl-C the second I see the message:

```
Press Control-C now to enter Aboot shell^C
Welcome to Aboot.
Aboot#
```

Looks great, but all is not well in the world of bootloaders. By using the `ifconfig` command, I can see that the `ma1` interface has no configuration:

```
Aboot# ifconfig ma1
ma1      Link encap:Ethernet  HWaddr 28:99:3A:BE:A0:B9
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
          Interrupt:36
```

I really hate when that happens.

WARNING

Watch out! Mashing on Ctrl-C too quickly will prevent the network configuration from loading in Aboot!

One of the tech reviewers for the first edition was one of the system developers, who agreed that "yeah, that's pretty lame." He also said that the NET statements were designed to get an image from the network while already in Aboot, and not on reboot the way I was using it. That makes sense, but I'm still a happy keyboard masher and always will be.

It's now 2019, and this still happens, but I've come to realize that my complaints about it are sort of dumb. You see, if you had configured `boot-config` to do this, why are you pressing Ctrl-C to go into Aboot? You would let Aboot do its thing and get the image from the network as intended, which I can assure you works just fine.

For years, I've trained myself to sit and stare at the screen and to pounce on Ctrl-C (or F2, or F12, or whatever damn key I've been commanded to engage) the millisecond that the message appeared. We've all missed those messages and had to reboot time and again, which is why this really drove me nuts.

NOTE

It's worth noting again that if your NET commands in the *boot-config* don't seem to be working, wait a few seconds after the *Press Control-C now to enter Aboot shell* prompt before mashing those keys. There's a good chance that they'll work if you can restrain your key-mashing impulse, if even for a few seconds. Better yet, just let Aboot do its job and stop interfering.

Honestly, doing this is a bit weird, so let me show you a more realistic way that you might use networking from within Aboot, and that is by manually setting an IP address and grabbing a file from the network using Linux commands:

```
Aboot# cd /mnt/flash
Aboot# ifconfig ma1 10.0.0.19
Aboot# wget http://10.0.0.100/EOS/EOS-4.20.1F.swi
Connecting to 10.0.0.100 (10.0.0.100:80)
EOS-4.20.1F.swi      100% |*****| 608M
00:00:00 ETA
Aboot#
```

Notice that I never configured a subnet mask? **ifconfig** will use the natural class mask if you don't specify one:

```
Aboot# ifconfig ma1
```

```
ma1      Link encap:Ethernet  HWaddr 28:99:3A:BE:9F:91
          inet  addr:10.0.0.1  Bcast:10.255.255.255  Mask:255.0.0.0
          inet6 addr: fe80::2a99:3aff:febe:9f91/64  Scope:Link
                 UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
                 RX packets:0  errors:0  dropped:0  overruns:0  frame:0
                 TX packets:6  errors:0  dropped:0  overruns:0  carrier:0
                 collisions:0  txqueuelen:1000
                 RX bytes:0 (0.0 B)  TX bytes:492 (492.0 B)
          Interrupt:36
```

To prevent this behavior, use the `netmask` keyword:

```
Aboot# ifconfig ma1 10.0.0.1 netmask 255.255.255.0
```

At this point, with the new version of EOS on board, we can edit the `boot-config`, put in the new `SWI=` statement, and reboot the switch, at which point it will load the new version of code. Here's the new `boot-config` after my vigorous editing using the `vi` editor:

```
Aboot# more boot-config
SWI=flash:/EOS-4.20.1F.swi
```

The switch will now boot the 4.20.1F code.

Booting most network devices into their bootloaders is an exercise in frustration, at least for me. I do it so rarely that I struggle to remember the arcane commands, and then I invariably look up what I need to do online. That process is made even worse by many networking vendors because different products boot differently. Aboot changes all that because the bootloader environment is just another flavor of Linux. If you find yourself in Aboot, just keep your wits about you and remember to look for the `boot-config` in `/mnt/flash`. Better yet? It's the exact same procedure on every Arista switch.

Speaking of which, let's talk about password recovery.

Password Recovery

On a competitor's switch, password recovery might be a nightmare with different arcane procedures and commands that vary depending on what model number device is being recovered. On Arista switches, every device is the same when it comes to password recovery. Think about it for a second: where is the *startup-config*? It's on *flash:*, of course. How do you get to flash in Linux? `cd /mnt/flash/`.

To do a password recovery on an Arista switch follow these steps:

1. Boot the switch.
2. Press Ctrl-C to enter Aboot.
3. From within Aboot change to the */mnt/flash* directory.
4. Edit the *startup-config* using vi.
5. Comment out the lines that begin with *username*.
6. Reboot the switch and log in as admin with no password.

That's it! And it's the same on every Arista switch. How cool is that? In a nutshell, if you know where the *startup-config* is and how to get there in Linux, you can password recover the box.

What if you loathe the vi editor? You can just rename the *startup-config* to *startup-config.bak*, after which, when you exit aboot, the switch will have a default configuration (note that this might trigger Zero-Touch Provisioning [ZTP]).

```
Aboot# cd /mnt/flash
Aboot# mv startup-config startup-config.bak
```

Conclusion

Running through Aboot once or twice is a good practice for anyone who works with Arista switches. I once mucked up a switch so severely that we couldn't get it to cancel or disable ZTP, and it wouldn't let us do any configuration. I was able to boot the switch, drop into Aboot, and issue the `fullrecover` command, which saved the switch, prevented us from bothering TAC (which saved a lot of time), and made me look like a hero. Let's face it, technical writers don't get all the girls because of our breathtaking vocabularies, so any chance to play the hero is welcome, even if this particular heroism was witnessed by only a bunch of IT guys in a cold data center.

Chapter 15. CloudVision

CloudVision is the name for a group of software products that allow control and management of Arista devices. Though designed to manage Arista devices, there's no reason that it couldn't support other vendors' devices through the application of a common interface such as *OpenConfig*, but for this chapter I'm going to include only Arista devices because it's a book about Arista, after all.

One of the reasons this book was delayed was because I struggled with CloudVision. When it first came out, I was not a fan, and with CloudVision being a big deal for Arista, I wanted to write about it in a positive light, but I just didn't believe in the product. That has changed in a big way, and the fact that it has is kind of a big deal to me. Allow me a brief digression as to why.

I was hired by Arista because of my ability to write about the company and its products (remember, I wrote the first edition when I was not an employee) and because I was clearly a fan. Arista's President and CEO Jayshree Ullal actually considered not hiring me directly because my independent views actually had a fair bit of impact in the market and she didn't want to lose that. I made a deal with the executives who hired me: the minute I stop believing, I'm gone. Additionally, I said that I would tell only what I felt to be the truth. If you've seen me speak, you know that I can be quite passionate about the topic at hand, but that passion evaporates if I have to tell marketing stories that vary

from the truth. Simply put, I can't lie to make a sale (I can, but I'm really quite bad at it), which has been both my greatest strength and my biggest failing depending on who you ask. To Arista's credit, that's kind of how it operates, as well, so I was hired.

When CloudVision first came out, it was far from perfect, and I could not in good conscience recommend it. That has changed and over the past year or two (this being 2019 as I write), CloudVision has improved dramatically in every way. Not only that, but it continues to improve both in functionality and in features. I do not necessarily recommend every feature for every customer, because every customer is different, but I do recommend the product now without reservation. I can absolutely say that there is something in CloudVision for everyone, even this old networking curmudgeon who will never give up his CLI access.

The CloudVision Family

There are two products currently that belong under the umbrella of CloudVision. They are CloudVision eXchange and CloudVision Portal.

CloudVision eXchange

CloudVision eXchange (CVX) is really nothing more than a vEOS instance running with more memory and processor than would normally be seen in a physical switch. [Chapter 21](#) presents an example of CVX in action, in which I use it as a controller of sorts for VTEP Flood-List automation. CVX does not require the other aspects of CloudVision (often abbreviated as CV, but here, I prefer to use the full

name) to work.

CloudVision Portal

CloudVision Portal (CVP) is the web-based tool that allows you to do an increasing variety of things to an Arista-based network in a graphical user interface (GUI) environment. You install CVP as a server, and you can install it in a cluster for resiliency, as well. You can order it as a standalone appliance, as well.

CVP is a pretty deep topic and is deserving of a large book unto itself, so there's no way I can cover all of its topics here. I will not cover how to install it or how to configure clustering or how to set it up because there just isn't room. With CVP being a graphical application, screenshots consume a lot of page space, so in a book that's about Arista, but primarily about EOS, there's only so much I can cover, especially when the desire is to cover it all.

Quick Things to Know

Without diving into how to install and configure everything in CVP, here are a couple of items that will help you in doing so.

- CVP really likes to work with Authentication, Authorization, and Accounting (AAA) set up because the username accessing the portal will be the username making changes to the switches. It can work without AAA, but it's happier with it enabled and working.
- The install image for CVP is quite large (more than 6 GB for version 2018.2.2), so be prepared for that if you have download access and want to try and run this on your laptop.

Keep the next bullet in mind, though.

- When installing CVP as a virtual machine (VM), the following minimum requirements should be met:
 - CPU: 16 cores
 - RAM: 22 GB (up to 250 devices)
 - Disk: 1 TB

As you can see, this is not really a “run it on your laptop” kind of application.

- If you’re running CVP on VMware, the CVP server does not support vMotion. Seriously, don’t vMotion the CVP server. You have been warned. Actually, as I understand things, if installed in a single server, the server can be vMotioned, but if installed as a cluster, it cannot. When in doubt, open a TAC case and ask because TAC is much better informed than I am and details like this can change over time.
- There are switch EOS version restrictions when using CVP. For example, version 2018.2.2 of CVP requires the switches it manages to be running at least EOS 4.17. That’s not an unfair requirement, because the current version at the time of this writing is 4.21, but just understand that this is a requirement. Also, if you have switches running versions of EOS older than 4.17, you really should upgrade them, anyway.
- You might need to install something called TerminAttr on the switches being monitored depending on the rev of EOS in use. It is included in EOS 4.17 and above, but you can always download the latest version as a SWIX and install/upgrade it using the EOS extension system or, as you’ll see, CVP itself.
- Technically the only supported browsers (as of 2018.2.2) are Firefox (v56+) and Chrome (v59+). I’ve used Safari, and it

works, but you'll get a "Safari is out of date" message. I can't speak for Internet Explorer, because, seriously, stop using Internet Explorer.

- eAPI needs to be enabled on the EOS devices if you're adding them manually. Discovered devices learned through Zero-Touch Provisioning (ZTP) will have this taken care of automatically.

NOTE

CVP has gone through some pretty significant changes over the years and as of this writing continues to do so. As a result, please accept my sincere apologies if some of the screenshots or sections I talk about seem out of place or have changed since this book was published. There are some very valid reasons for such changes that are far (far!) outside the scope of this book, but so far as I've seen, the changes have been either powerful backend changes or slight user interface changes such as moving sections around or changing the first page a user sees when they log in. The fundamental principles generally remain the same, though, so hopefully the impact of any discrepancies should be small.

Sometimes, book deadlines and software releases overlap in a way that makes me unable to make all the images match, and I assure you that I have lost the appropriate amount of sleep obsessing over this fact.

As always, check the release notes before installing any software because these requirements can change as versions progress.

CloudVision Portal

When people say CloudVision, they generally mean CVP. At Arista, you might hear people refer to the suite of products as CV, but even then, I've heard people refer to CVP as CV, so don't get too caught up in semantics.

What CVP Is

When CVP was first released, it was primarily a tool that allowed the management of device configurations (including initial and replacement configuration) in an automated way that does not require scripting or programming knowledge. That's not to say that you can't script CVP, because you can, but you don't need to, and that flexibility is a hallmark of Arista that you'll find almost everywhere. This configuration management aspect of CVP is called *Network Provisioning*.

As CVP grew it gained additional capabilities including, but not limited to, workflow automation, State Streaming Telemetry, Network-Wide Rollback, Compliance, and so on. CVP also has the ability to integrate with other vendors' products such as those from ServiceNow, VMware, Check Point, and PaloAlto, just to name a few, and CVP apps can be added to further expand the functionality.

What CVP Isn't

CVP is not a traditional Simple Network Management Protocol (SNMP)-based network-monitoring tool. Though you might be quick to compare it to old-school tools like HP OpenView or Solarwinds given a quick look at the GUI, that's not what CVP is about. CVP is different. How? Hopefully that will become clear as you read on, but the biggest thing in my opinion is that it does not use SNMP.

Some of the older marketing material talked about CVP being a *single pane of glass* into the network, a description that Arista has rightfully moved away from, in my opinion. Honestly, if the company had been

the first to come up with that term, it might have worked, but so many products have promised to be a *single pane of glass* that it's developed a reputation that's sure to garner eye-rolls from anyone who's been through one of those sales meetings from any vendor.

CVP gives a serious amount of visibility into your network, unlike any other tool that I've ever seen, and it deserves to be checked out for that reason alone.

How CVP Works

There are a few aspects of CVP that bear discussion of the underlying mechanisms by which they work.

CONFIGURATION MANAGEMENT

Configuration changes to switches are done through eAPI ([Chapter 30](#)) and a form of the `configure session` capability of EOS ([Chapter 9](#)). Believe it or not, even if only a single line of configuration is changed through CVP, the entire configuration of the switch is written each time through the use of configuration sessions. This isn't some lame "use SSH to simulate a user applying a command" type of solution. This is using the EOS tools to accomplish something better.

STREAMING TELEMETRY

Streaming Telemetry is accomplished through a relatively recent addition to EOS called *TerminAttr* (pronounced term-in-ate-er). Because the minimum version of EOS that ships with TerminAttr installed is 4.17, if you're running 4.17.x or later, TerminAttr can be installed or upgraded as an extension ([Chapter 16](#)) or through CVP

itself. If you’re running a version prior to 4.17.x, you should really upgrade, but you can still install TerminAttr the same way. With TerminAttr streaming to CVP, the live state of the switch is streamed from SysDB to the CVP database in real time. Remember when we used to poll SNMP every five minutes? Well, CVP telemetry reports in real time, and it includes everything you could possibly care about from every device being monitored without you having to figure out things like interface indexes and the object identifiers (OID) of some obscure bit of data. Did I mention that it’s in real time, because it’s in real freaking time. As someone who used to build network operations centers (NOCs) for a living and who spent way too much time configuring SNMP collection, the real-time aspect of Streaming Telemetry blows my mind.

TOPOLOGY

Through a combination of Link Layer Discovery Protocol (LLDP) discovery and user assignment of device roles (I cover this later in this chapter), we can build a graphical view of the network topology.

CVP APPS

CVP can also have CloudVision applications added to it, which means that you can expand it to do things outside the original scope of the design. Integration with ServiceNow is one such application that allows change controls to integrate with that vendor’s solution. I hate to say things like the possibilities are endless, so how about the possibilities are limited only by your imagination? No, that’s awful, too. Look, there are a lot of possibilities.

Let's look at some screenshots.

Login Page

I just had to include a screen capture of the wonderful new CloudVision login page because, as you can see in [Figure 15-1](#), it has clouds. Although that in and of itself might not be all that exciting, the old one was a stock photo of a data-center rack that was nice and all, but not befitting the CloudVision name. Now there are clouds!

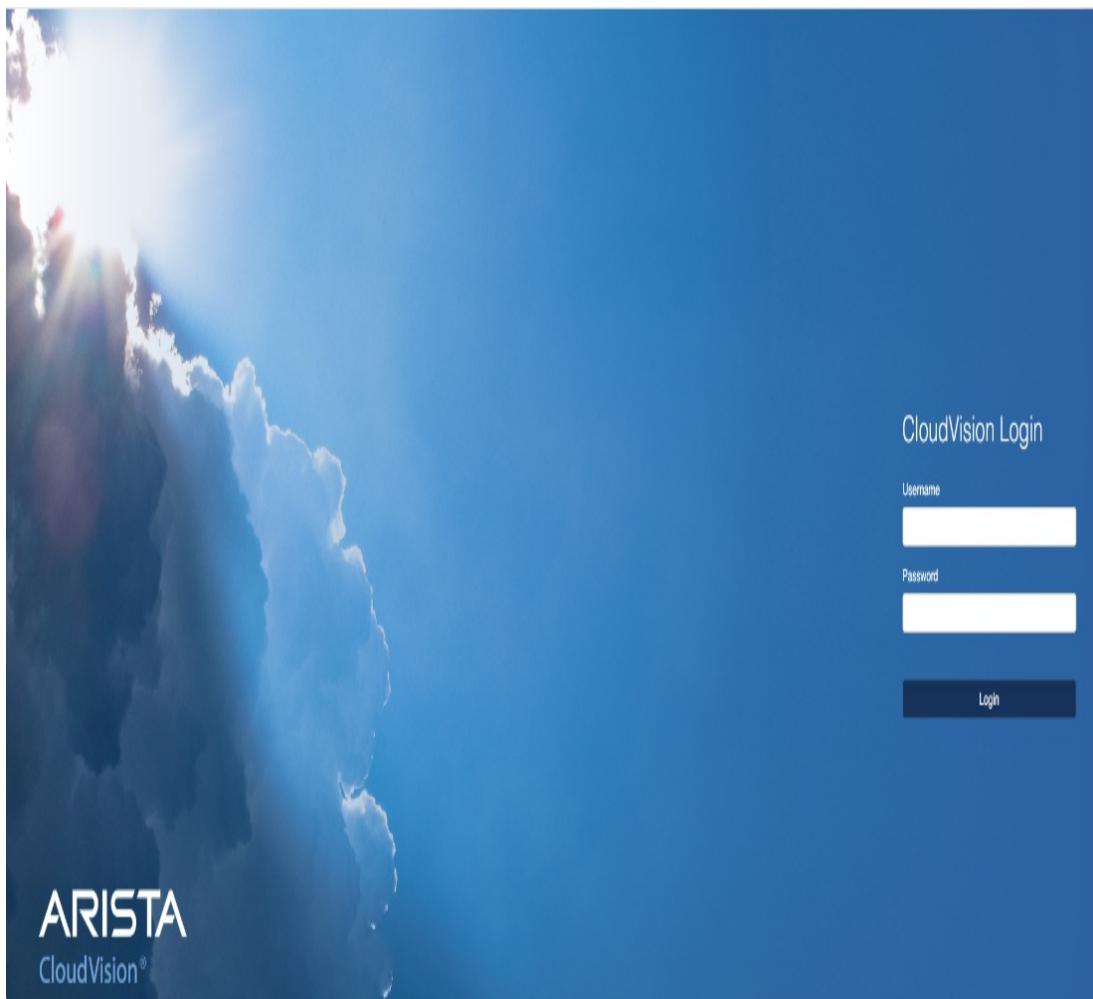


Figure 15-1. The CVP login page

That's all. I just wanted to highlight the new login page because it's so

pretty. Call me shallow, but I'm a man who appreciates a sexy login page. The page that opens after you log in varies on different revs of CVP, so I'm going to just show some examples from the most commonly used sections in the order in which they appear in the menu for the latest version at my disposal.

Devices

The first page you'll probably find yourself in with an existing CVP installation is the Inventory page on the Devices tab, an example of which is shown in [Figure 15-2](#).

All Devices > Inventory								
Inventory	Showing all 16 devices						Add Device	
Compliance	Device	Status	Model	Software	TerminAttr	IP Address	MAC Address	Serial Number
	agra	✓ 🚧	7280SR24-48YC6	4.21.0F	1.5.4	172.22.8.35	28:99:3a:a5:d6:85	JPE17191574
	bugalerts-demo	✓	vEOS	4.20.11M	1.5.4	10.92.48.193	52:54:00:c2:2a:ab	2A814882069E44FD854C2...
	cv-demo-sw1	✓	7150S-24	4.21.3F	1.5.2	10.92.48.14	00:1c:73:1e:7b:04	JPE12233288
	cv-demo-sw2	✓ 🚧	7150S-24-CL	4.21.2F	1.5.4	10.92.48.15	00:1c:73:2b:1d:1c	JPE1330030
	cv-demo-sw3	✓ 🚧	7150S-24	4.21.2F	1.5.4	10.92.48.16	00:1c:73:00:43:7c	JAS12110003
	cv-demo-sw4	✓	7150S-24	4.21.3F	1.5.2	10.92.48.17	00:1c:73:63:ce:e9	JPE14413861
	LF1	✓ 🚧	7280SR-48C6-M	4.21.3F	1.5.4	172.16.0.1	44:4c:a8:73:b1:bb	JPE16200889
	LF2	✓ 🚧	7050SX-72Q	4.17.5M	1.4.1	10.92.48.2	44:4c:a8:24:97:81	JPE16012748
	LF7	✓ 🚧	7050SX-72Q	4.17.5M	1.5.4	10.92.48.7	44:4c:a8:24:88:21	JPE16012645
	LF10	✓ 🚧	7050S-64	4.18.10M	1.4.1	10.92.48.10	00:1c:73:4e:81:35	JPE13403596
	LF12	✓ 🚧	7050S-64	4.18.10M	1.4.1	10.92.48.12	00:1c:73:4e:d3:17	JPE13414749
	mumbai-eos	✓ 🚧	Redstone-XP-02060	4.21.0F	1.4.1	172.20.252.74	00:90:ec:38:3c:9b	D2060B2F125731A9000016
	SP1	✓ 🚧	7050QX-32S	4.21.1F	1.4.1	10.92.48.201	44:4c:a8:88:a7:ab	JPE18071095
	SP2	✓ 🚧	7050QX-32S	4.21.1F	1.4.1	10.92.48.202	44:4c:a8:88:7a:fb	JPE18071399
	vEOS-d5-TFD1	✓ 🚧	vEOS	4.18.0FX-vEOS.C1...	1.4.1	10.92.48.59	00:50:56:25:26:8d	F2F30C2D31A194DBFC30...
	vEOS-d5-TFD2	✓ 🚧	vEOS	4.18.0FX-vEOS.C1...	1.4.1	10.92.48.58	00:50:56:65:aa:8d	908ED47CDF6992ABAB0F...

Figure 15-2. Device inventory in CVP

On the Device Inventory page, you can select individual devices to view detailed information about each, from the *running-config* right down to current real-time interface counters, much of which was formerly found in what Arista originally called *Telemetry*. Figure 15-3

shows an example of a device selected by clicking the Processes link on the left side, which shows real-time scalable graphs.

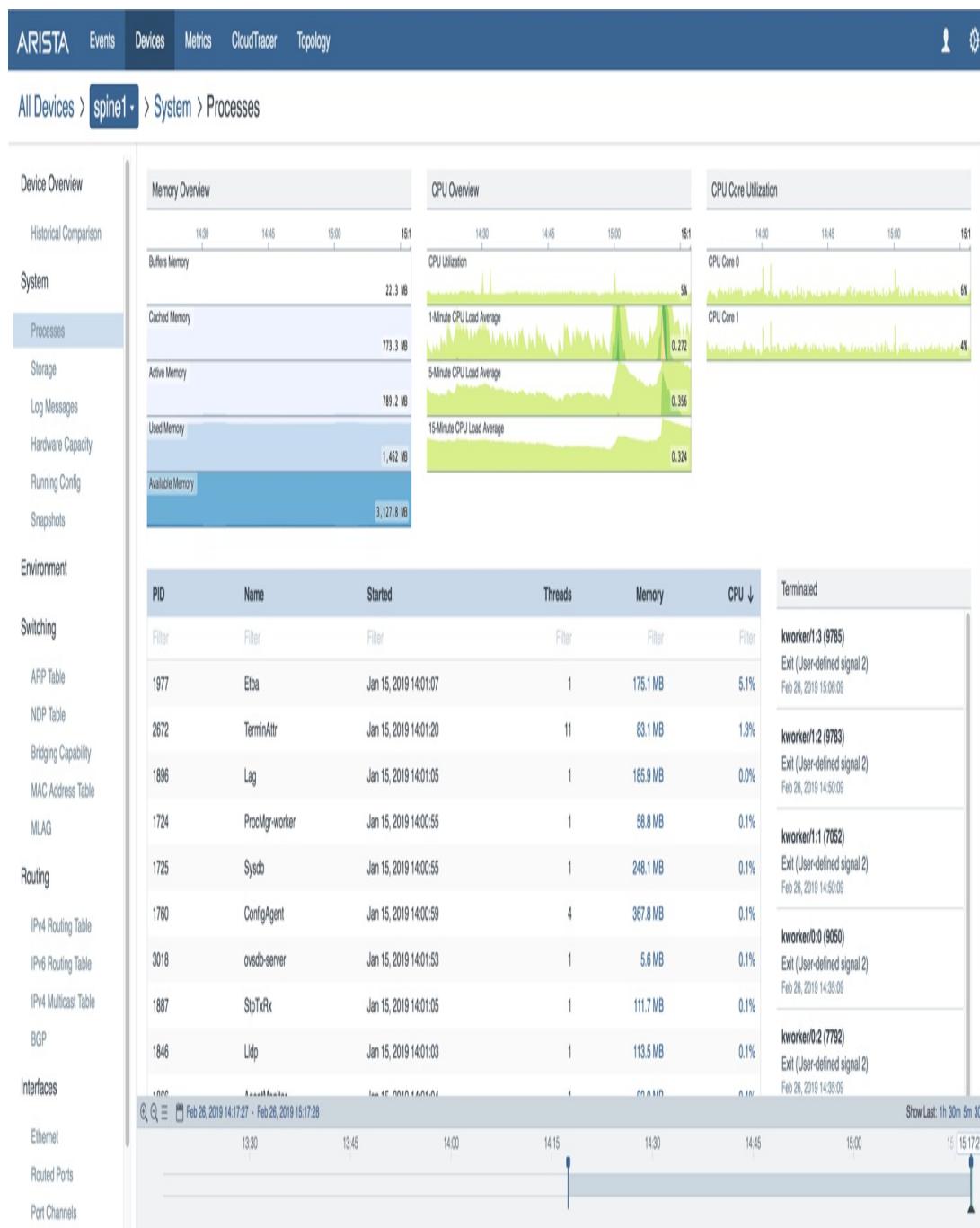


Figure 15-3. CVP device inventory CPU stats

The other option on the Devices tab is Compliance. The Compliance

page shows some terrific information such as whether your installed versions of EOS have outstanding bugs or security alerts, as shown in Figure 15-4. It is also worth mentioning that CVP will alert you to bugs through a feature called Bug Alerts and that Arista has a bug portal on its website so that the customers can look up bugs. That portal is right on the Arista.com support page and requires a customer login. This is just one more example of the openness that Arista seems to embrace.

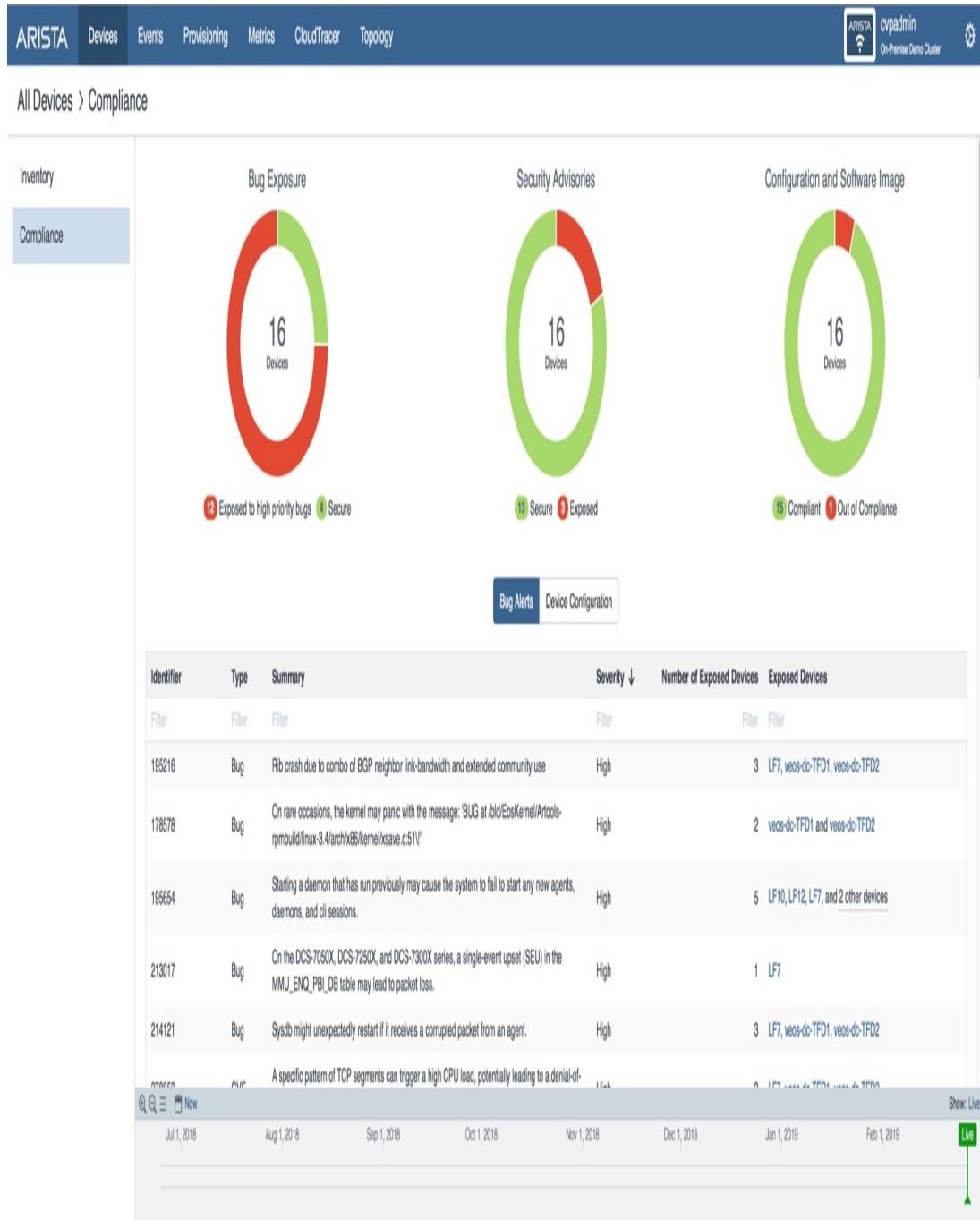


Figure 15-4. Device compliance in CVP

In previous versions of CloudVision, CVX was necessary for this functionality, but this has been folded into CVP in early 2019 so that the installation can be simpler for those who don't otherwise require CVX.

Looking at pages like this, it's easy to see why Arista is so tempted to use the *single pane of glass* description. Hell, this is only the tip of the iceberg of what CVP can do, but perception trumps reality, and the sad truth is that other vendors long ago ruined the phrase.

Though I've shown only a couple of screenshots, both of these pages are pretty darn powerful. The fact that I can look at a single pane of...
<grumble>...a single page that shows me any and all outstanding security and bug alerts for all of my devices is a pretty nice feature, and you really must play around with the real-time telemetry page to truly appreciate it. Yes, the graphs are pretty, but as soon as you begin getting used to the real-time nature of this tool, you will be forever ruined for other vendors' tools.

Network Provisioning

I like to describe Network Provisioning in CVP as a hierarchical network configuration tool. Basically, devices are placed into containers, and you can place containers into other containers in a hierarchy, as shown in [Figure 15-5](#). Thus, it's a hierarchical configuration tool. See? I don't just make stuff up! Well, I totally made that up, but it's logical, so I'm keeping it.

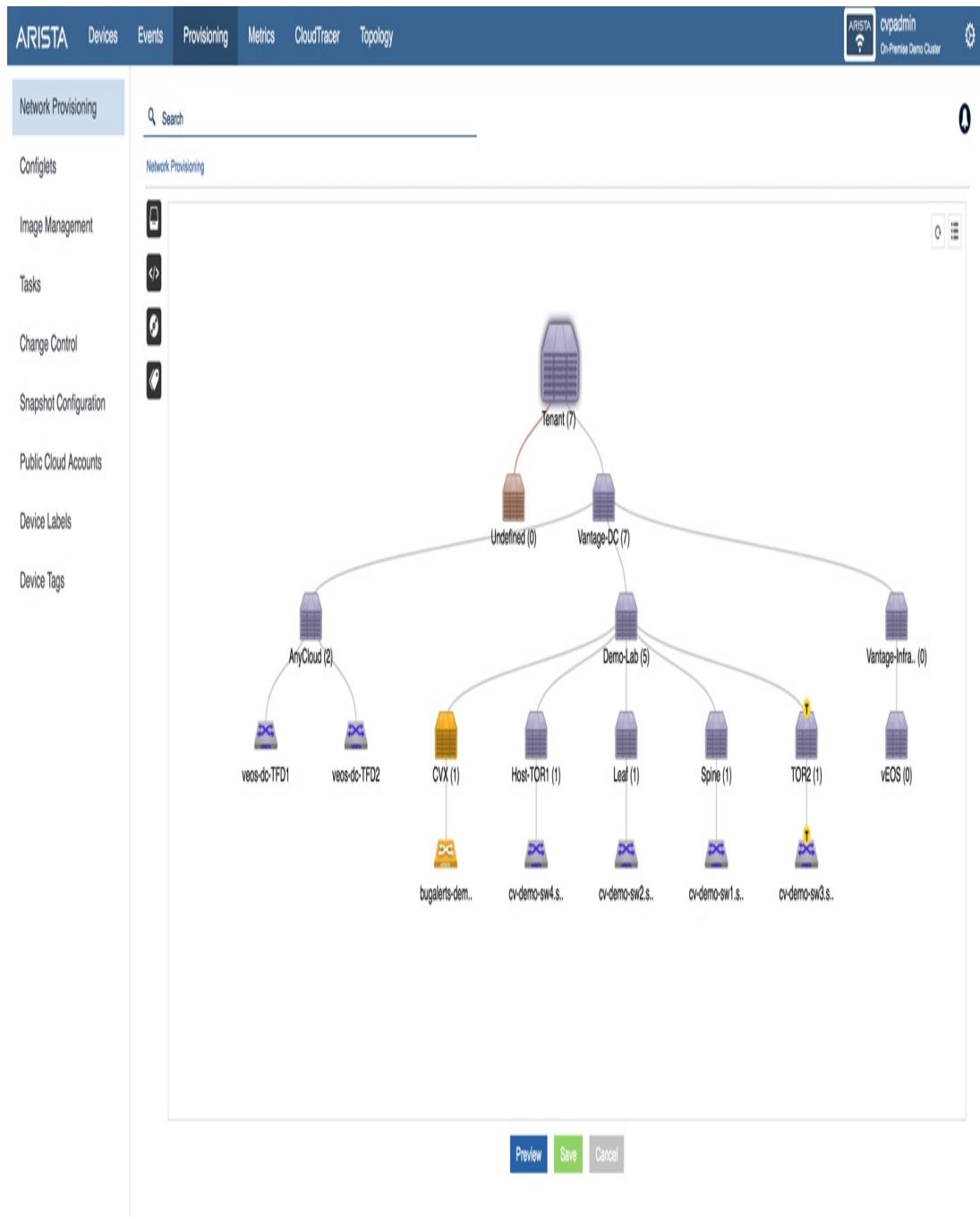


Figure 15-5. The Network Provisioning page

There is a root container named *Tenant*, under which all other devices and containers are located. If you look at the *Tenant* container in Figure 15-5, you'll see that it's labeled *Tenant (7)*. The seven in parentheses is not the number of objects within the container, and it's

not the number of containers within the container; rather, it's the number of nodes (generally switches) total within that container, including all nodes in all subcontainers. And, yes, you can rename the Tenant container.

NOTE

When using CVP, you can zoom into any section of the hierarchy with the scroll wheel on your mouse. Double-clicking a container shows its included devices on the left side of the window, whereas double-clicking the name of the container allows you to rename it.

The way to think of containers is, again, as a hierarchy, but what's cool about this is the fact that it's a configuration hierarchy. Arista accomplishes this through something called *configlets*. Configlets are pieces of a configuration that are common to the container and I'll show them in more detail in a minute. Think about all of your network device configurations. Assuming that they're all EOS devices and further assuming that you're using TACACS+ or Radius, they probably all have the same AAA configuration commands. They might all have the same useful aliases configured. They might all have the same eAPI configuration, as well. By creating configlets with those common CLI commands and applying those configlets to a container, every device within that container will have the same configuration applied, but only from the applied configlets.

Now, think for a minute that you have a subset of devices that are all spine switches. Those spine switches might have common configurations that are not shared by leaf switches. Similarly, the leaf

switches have configurations that are common to one another but are not shared by the spines. By putting the leaf switches into a container and the spine switches into a different container, this can be accomplished with ease.

It is important to understand that the view shown in the Network Provisioning section is not a network topology, but a configuration hierarchy. That comes as second nature after using CVP for a while, but it can confuse people who aren't used to the tool. CVP does have a topology view, which I cover in a bit.

How does CVP learn about a device? It can be done manually, which is boring, or it can be learned dynamically, which is cool and exciting. Can you guess which one I'm going to cover?

If you've read about ZTP ([Chapter 13](#)), you know how powerful a tool it can be. CVP can actually use Arista switches' default ZTP capabilities to its benefit by becoming a ZTP server. To allow this, you must first enable Dynamic Host Configuration Protocol (DHCP), which is outside the scope of this chapter (you can also use a separate DHCP server), but as soon as the CVP server is aware of new DHCP-configured devices, it can act as the ZTP server.

Switches learned through the ZTP server are put into the Undefined container, which is attached to the Tenant root container, as illustrated in [Figure 15-6](#).

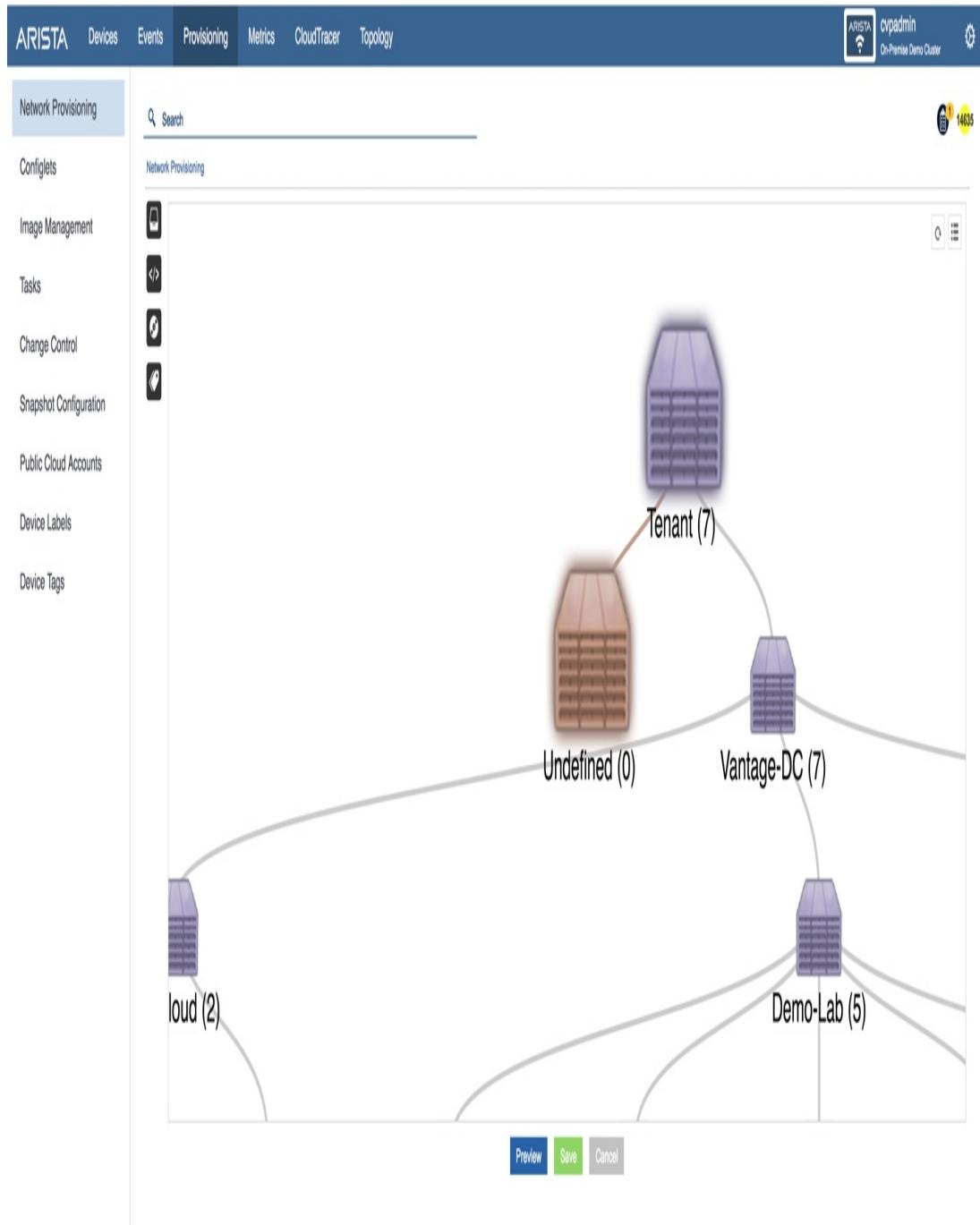


Figure 15-6. Undefined container

A ZTP-discovered switch is added to the undefined container, which means it will have whatever configuration the Tenant container has associated with it, but no others. Move the device to another container and that container's configlets will be applied. To do so, right-click the

device and choose Move.

A QUICK MESSAGE ABOUT NETWORK PROVISIONING

Here's the message: you don't have to use it. Although this is a very powerful capability, some networks just aren't built in such a hierarchical fashion. Should they be? That's not for me to decide, because it's not my network. There are options, though, and Arista loves to give you options.

First, you don't need to use Network Provisioning at all. You can configure all of your switches manually, like you've always done, or you can use another tool like Chef or Puppet or Ansible. Personally, I'm a fan of anything hierarchical if for no other reason than I get to walk around saying *hierarchical* all day. Really, though, I'm a fan of logical designs, and if you can design your network to take advantage of such a tool, that's great! The simple truth is that a lot of networks just might not fit into this model today, and that's fine because CloudVision has a bunch of other tools that are so freaking useful it's difficult to ignore them.

CONFIGLETS

A configlet is a snippet of configuration that can be applied to every device within a container or to individual devices as desired. Because of the latter capability and the ZTP-learning capability for new devices, it is possible to completely configure a network without actually logging into a switch through CVP. Indeed, I have seen customers who have designed new networks in such a way as to keep network engineers from needing to use CLI, instead relying on the use of CVP

to manage the devices. Personally, I think that's a tad excessive, but then I've always been a CLI control-freak kind of guy. I also acknowledge that some network managers might well want to keep CLI control freaks away from their switches.

Back to the hierarchical model, the configlet in [Figure 15-7](#) shows an example of a CVX configuration that would be applied to all switches (nodes, technically) within a container. Of course, if you had different parts of the network with different AAA requirements, you could have different configlets assigned to different containers as needed.

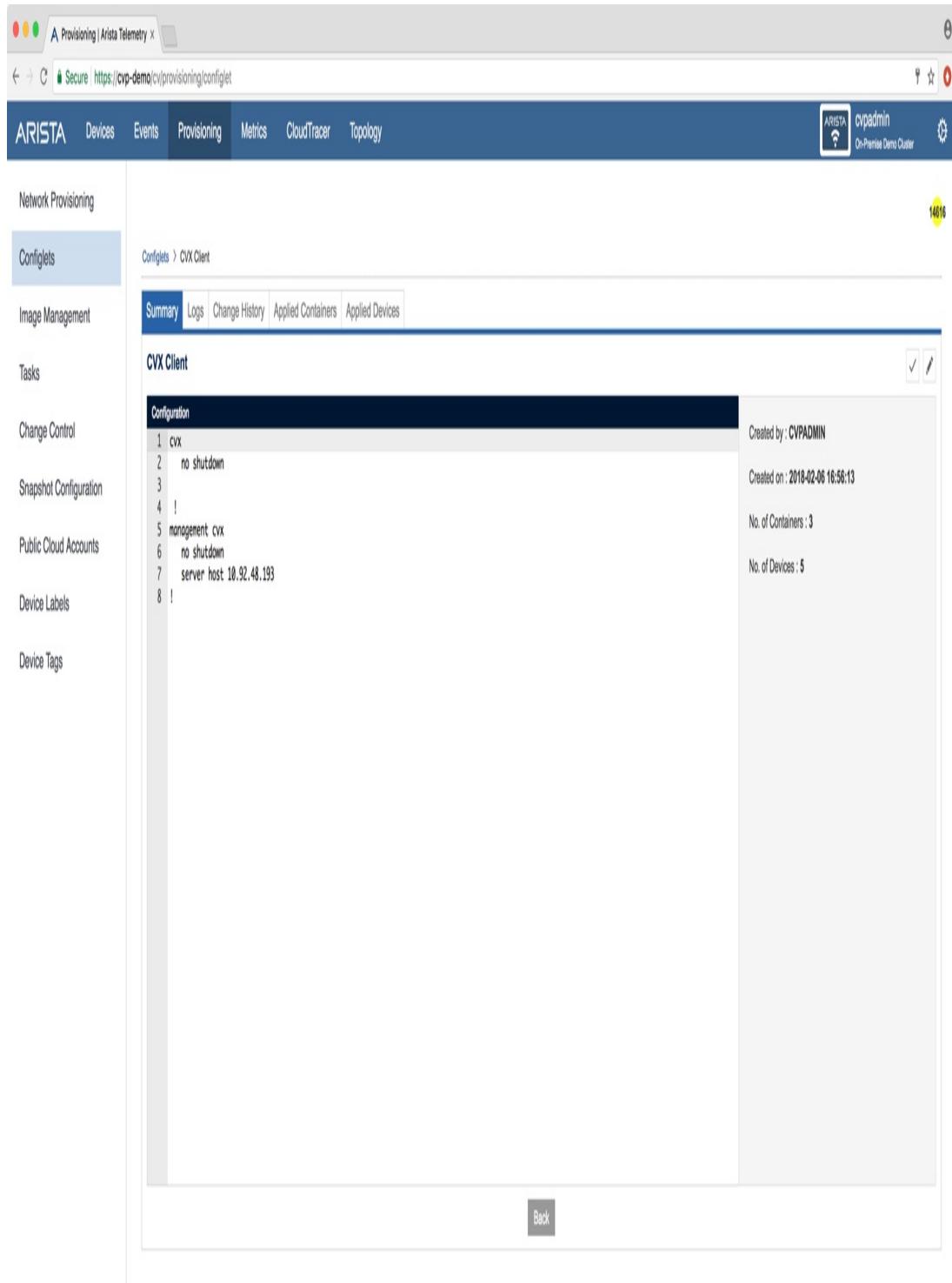


Figure 15-7. A configlet

The cool thing about almost everything Arista related is that there are options, and as I wrote earlier, you don't even need to use the

configuration management part of CVP. As you’re about to read, there are a lot of other very powerful aspects of CVP that many people are drawn to that can be used alongside of other configuration management tools, such as Ansible.

CONFIGLET BUILDER

Configlets don’t need to be static. They can actually be built dynamically using Python scripting or even a webform-based tool, both of which can be built using the *configlet builder*. When adding a configlet, you can write (or more likely paste) one from scratch, or you can use the configlet builder tool to help. [Figure 15-8](#) depicts the configlet builder showing a script from a lab environment.

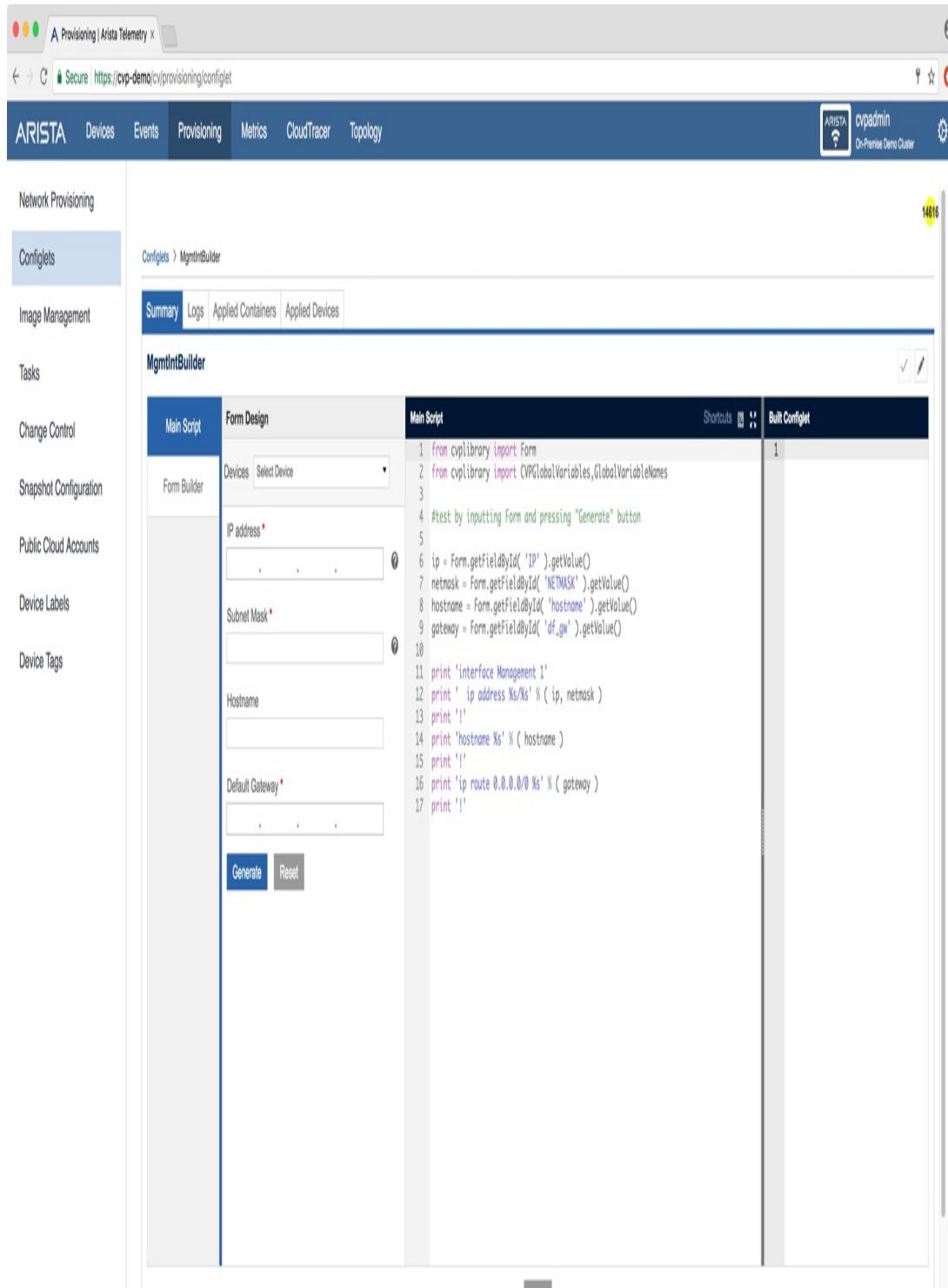


Figure 15-8. Configlet builder

The configlet builder is a very powerful tool that is far beyond the scope of this chapter, but the sample in Figure 15-8 is a nice example

that shows how you can configure IP addresses by filling in a form on the CVP configlet builder tool. These configlets can be as small as one line of CLI configuration or thousands of lines of Python code. That's the beauty of the system: you're not tied into either. The system is designed such that it's up to you how to use it.

TASKS

A *task* is something that needs to be done or has been done. You know, like a task in real life. Tasks are what get done during change controls (which we discuss in the next section) and can be configuration changes, image pushes, or anything else that you might think of as a job to be done. [Figure 15-9](#) shows an example task highlighting the configuration change (the changing of the IP address in the `ip name-server` command).

Figure 15-9. A CVP task

CVP keeps tasks available for viewing after they've been run, and you can see log entries associated with the tasks by selecting that tab on the

top of the task.

You don't really create a task directly. Instead, suppose that you right-click a node or container and apply a configlet. When you do that, you change the status of the selected object and, when saved, they will update on the Network Provisioning page, displaying a yellow T, as shown in [Figure 15-10](#). At this point CVP has created a separate task per node within the container.

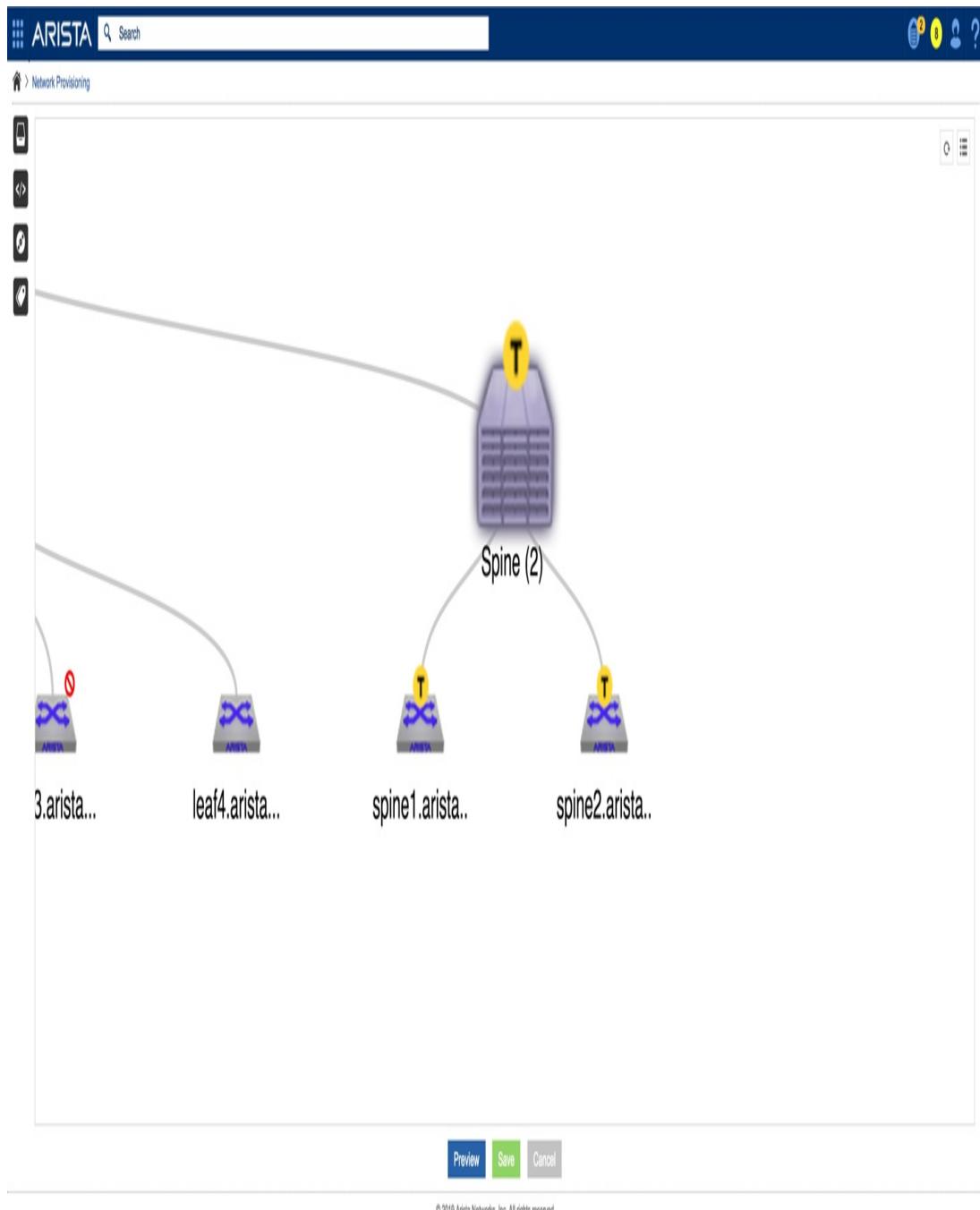


Figure 15-10. Pending tasks in Network Provisioning

In this case, I had right-clicked the Spine container and applied a configlet to it. That change isn't committed, though, until I either run the task manually or process a *change control*.

CHANGE CONTROL

Change control is where tasks are committed to nodes or group of nodes. In [Figure 15-11](#), I've created a change control with which I'm pushing a configlet to a container. This page updates every few seconds to show continued status updates while the task is running. Change controls allow users to push individual or groups of tasks that have been created. You can run them in a specific order, if desired, and push them automatically based on a schedule.

CVP takes a snapshot (see the next section) of the node before and after the change because it's always a good idea to do so.

Task ID	Name	IP Address	Model	Executed On	Executed By	Snapshot Template	Order	Snapshot	Snapshot Status	Status
1203	cv-demo-ev3.sjc.arista...	10.92.48.16				Snapshot - Maintenance	1	View	Pre-Snapshot completed	Pending

Figure 15-11. Change control being pushed in CVP

When the change control completes, the pending tasks will also be completed, and the yellow Ts will disappear from the Network

Provisioning page.

You may notice a slight difference in the layout of this page in my examples, which was sourced from a slightly older version of CloudVision. The biggest cosmetic difference is that the tabs (Devices, Events, Provisioning, etc.) are not there on the top, instead showing a small Rubik's Cube-looking square of squares that served as the main navigation for the older versions.

If you're curious why I would include a screenshot from an older version, I could say something like I wanted to show an example of the older interface, but the simple truth is that it's quite difficult to build a suitably impressive CVP environment due to the sheer number of devices (or VMs) required to do so, and this shot is from one of the demo solutions available to me at Arista, whereas the other screenshots are from a different demonstration solution. Also, I already had the screenshot, and re-creating it in a new version seemed like real work when I had already enjoyed the sensation of missing about 38 deadlines.

Back to the example in [Figure 15-11](#), if you look above and below the progress bar, you can see Pre Snapshot and Post Snapshot sections. That's a great way to segue into snapshots.

SNAPSHOTS

A *snapshot* is a collection of data about one or more devices at a single point in time. By default, a snapshot is more than just the running configuration, and a snapshot can be taken for a node or a container. Additionally, you can create custom snapshots to record whatever you

want. [Figure 15-12](#) shows an example Snapshot Configuration screen in CVP, and you can see the example selected showing a custom snapshot that's recording the output of the `show ip bgp` command.

Name	Commands	Devices	Status	Actions
Snapshot 7	2	foxtrot and golf	Valid	
Snapshot 14	2	bravo, charlie, hotel, and 1 other device	Pending	
Snapshot 17	1	foxtrot	Pending	
Snapshot 28	1	delta, hotel, india	Valid	
Snapshot 28	1	charlie and india	Pending	
Snapshot 34	2	golf, hotel, india	Valid	
Snapshot 34	2	bravo, charlie, delta, and 3 other devices	Valid	
Snapshot 42	3	india	Invalid	
Snapshot 48	3	foxtrot	Invalid	
Snapshot 58	3	foxtrot and golf	Valid	
Snapshot 62	3	hotel	Pending	
Snapshot 65	2	bravo, charlie, device-with-a-much-longer-name, and 4 other devices	Invalid	
Snapshot 84	3	charlie and india	Valid	
Snapshot 93	1	foxtrot	Valid	
Snapshot 95	3	charlie, delta, device-with-a-much-longer-name, and 3 other devices	Valid	
Snapshot 95	1	charlie, delta, india	Pending	
Snapshot 96	1	charlie, device-with-a-much-longer-name	Deadline	

Figure 15-12. Snapshots

Think about how you might have done change controls in the past. Have you ever issued a `show` command, performed the change, and then did the same `show` command to see the results of the change? Of course you have! CVP allows that to be automated, and between snapshots and the fact that they are executed before and after change control submits a task, CVP has a wealth of current and historical information that can be incredibly useful, especially when the need for rolling back changes occurs. You can even compare between the pre- and post-change snapshots.

IMAGE MANAGEMENT

Part of CVP's power lies in the ability to manage images such as the EOS SWI images used on Arista switches. [Figure 15-13](#) presents an example of the Image Management page.

Name	Containers	Devices	Notes	Uploaded by	Uploaded Date
EOS-4.21.3F-2GB	0	2	Add Note	cvuser	2019-02-25 04:40:02
4.17.5M + terminaltr1.5.4.1	0	0	Add Note	cvuser	2019-02-11 05:10:07
EOS-4.20.11.M.swi	0	0	Add Note	cvuser	2019-02-11 04:57:20
EOS-4.20.11M	1	1	Add Note	cvp system	2019-02-11 19:45:50
EOS-4.20.7M	1	0	Add Note	cvuser	2019-01-30 19:08:39
EOS-4.21.2F-2GB	1	2	Add Note	cvadmin	2019-01-30 19:08:14
TestBundle	0	0	Testing AB	a	2018-11-27 15:35:02
EOS-4.20.8M.swi	0	0	Add Note	cvuser	2018-08-20 16:38:45
EOS-4.17.8M	2	0	Add Note	cvadmin	2018-08-20 11:53:45
EOS-4.20.5F	0	0	Add Note	cvadmin	2018-08-20 11:53:25
EOS 4.18.4 + Terminaltr	0	0	Add Note	cvadmin	2018-08-20 11:53:03
EOS-4.18.8M	0	0	Add Note	cvp system	2018-08-27 13:46:37

Figure 15-13. CVP image management

This is not really all that complicated, but something that can be confusing is that the *images* you see on the Image Management page are not necessarily a single image; they might be a collection of images to be installed as a set called an *image bundle*. What makes this

confusing is both the name *images* and the fact that a lot of people name the images the same as the software image contained within them. If that makes your head spin, consider the detail of the image named EOS-4.21.2F-2GB shown in [Figure 15-14](#).

In this CVP image, there are two software images: EOS-4.21.2F-2GB.swi and TerminAttr-1.5.4-1.swix. Even though this is perfectly reasonable from a logistical point of view (I want to make sure that version 1.5.4-1 of TerminAttr is always installed when EOS 4.21.2F-2GB is deployed), I find the use of the term *image* to describe a collection of images confusing. Just remember that when you push an image (using a Task submitted through a Change Control), the image you're pushing might, in fact, be multiple images (an image bundle).

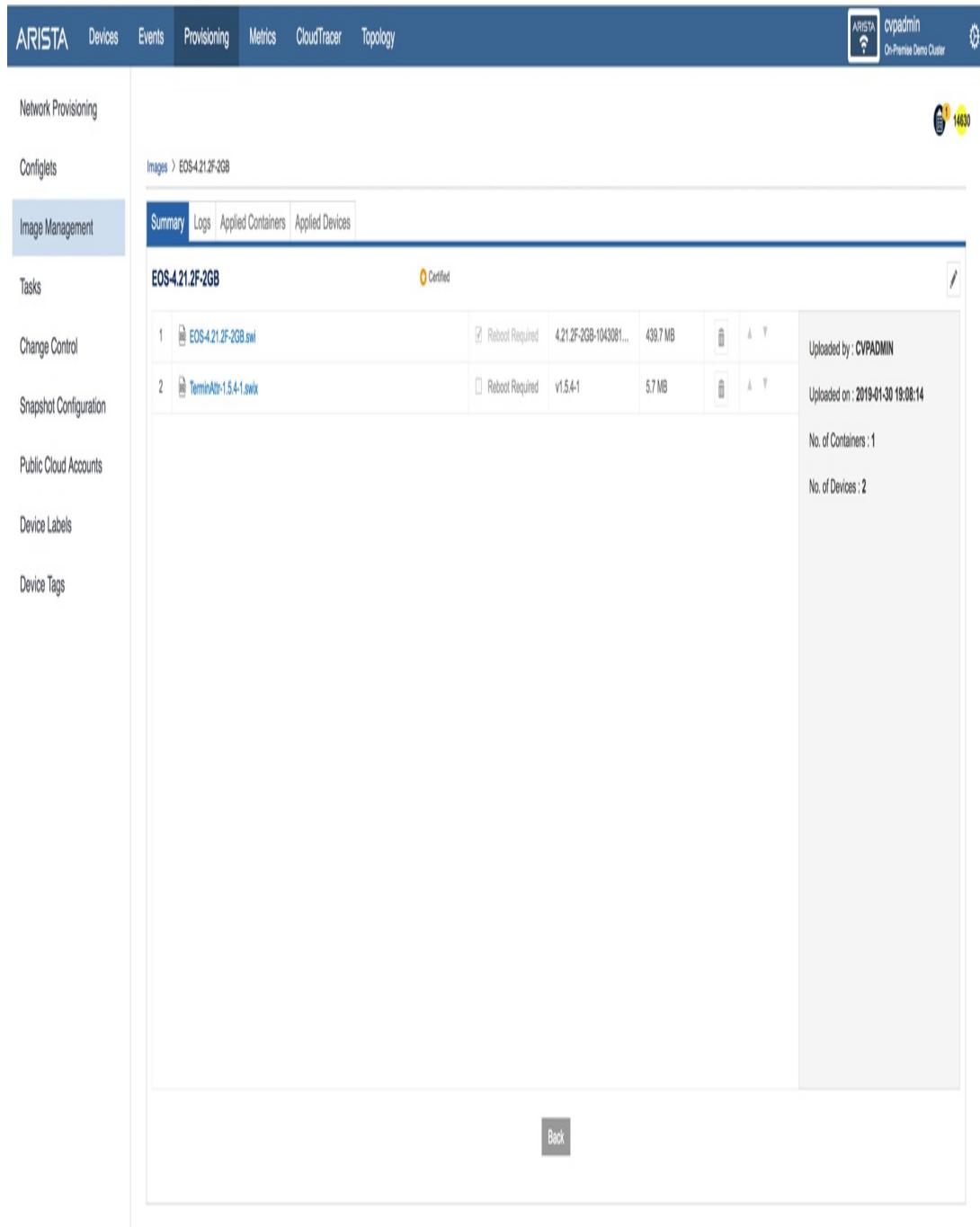


Figure 15-14. Detail of a CVP image

One of the useful parts of the Image Management page is the fact that you can see what containers or devices an image is applied to. Honestly, the fact that you can bundle software images together is very powerful. I just wish it was called something else because I'm a

pedantic pain in the ass.

ROLLBACK

Rollback is the ability to revert to not only a previous configuration, but also a previous image. You can roll back a device or a container, and remember that a container can contain other containers, so the flexibility is pretty significant.

There are two major types of rollback: *device* and *network*. *Device rollbacks* are pretty obviously named as they affect only one device, but *network rollbacks* aren't really network-specific but rather container-specific.

You roll back by right-clicking a node on the Network Provisioning page. Then, on the menu that appears, choose Manage, and then click Rollback. Figure 15-15 shows a device rollback for the node named veos-dc-TFD2. The icon for selected object on the Network Provisioning page is a bit larger than other objects and has a subtle glow effect applied to it. The right-click menus appear where there is room for them like most contextual menus that you're used to seeing.

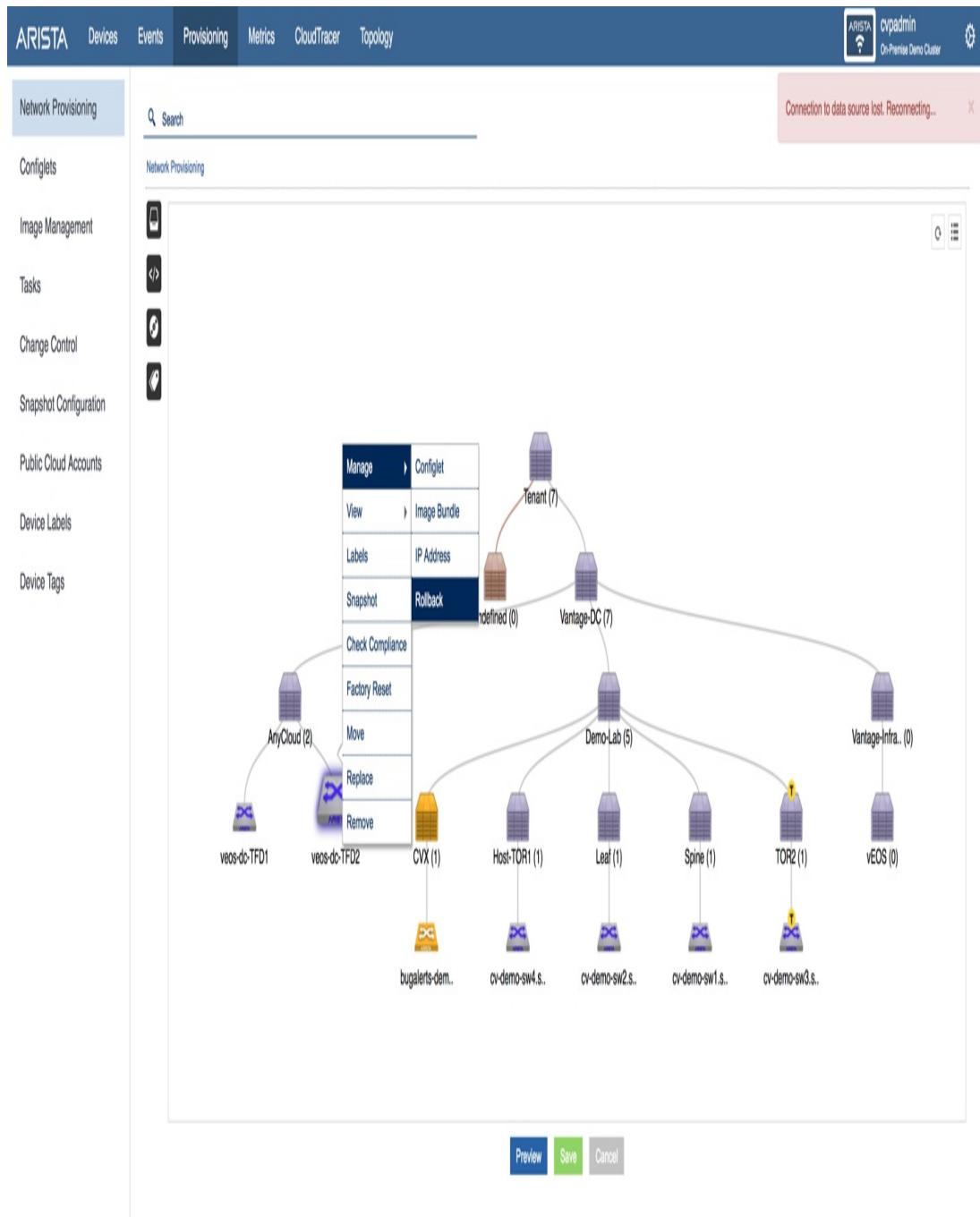


Figure 15-15. CVP device rollback

If you right-click a container and follow the same steps, you'll see the option for Network Rollback instead of simply Rollback, like you did for a device. [Figure 15-16](#) shows an example of this in which I have selected the main tenant container.

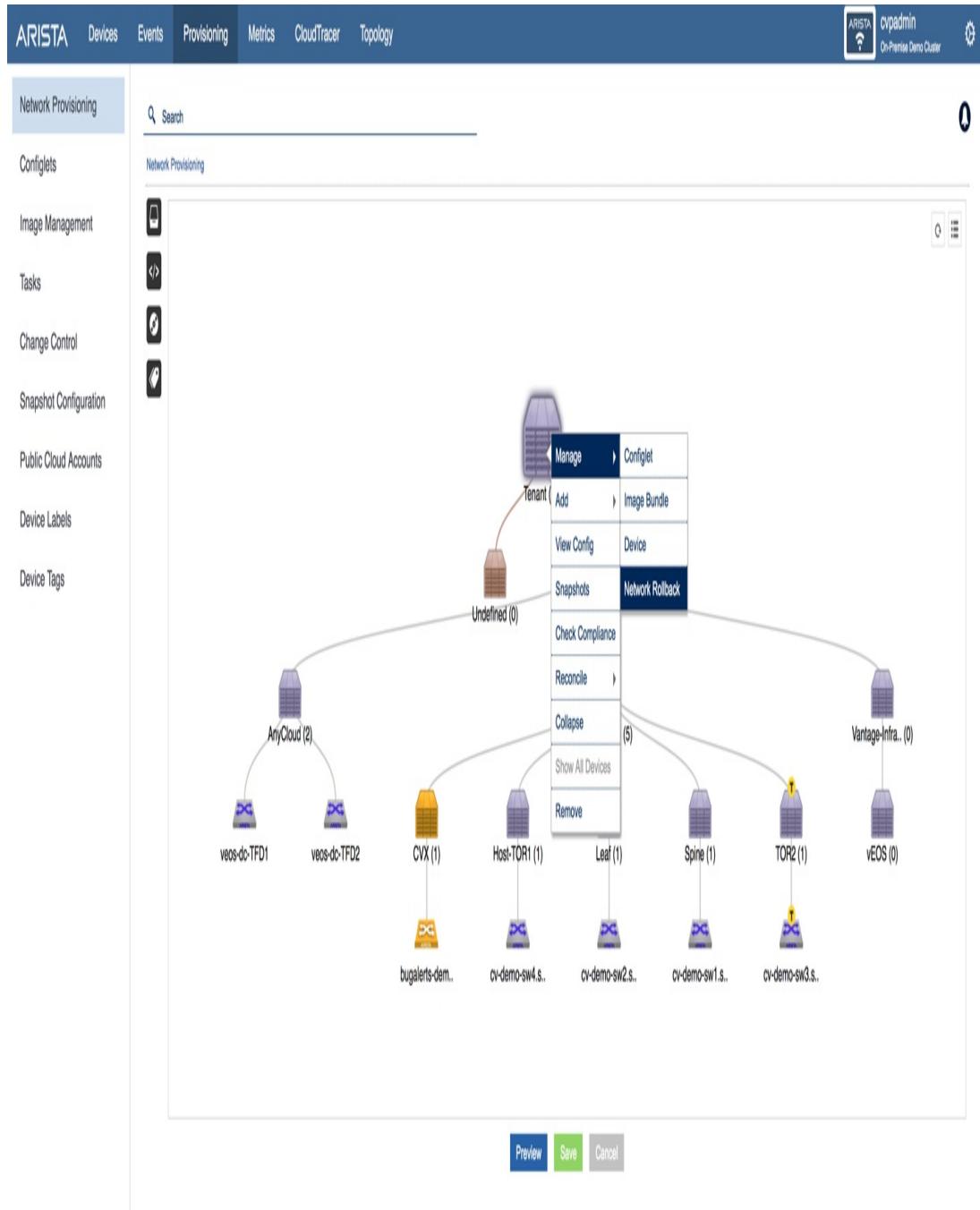


Figure 15-16. CVP network rollback

With either of these selected, you are then taken to the rollback page, which shows you the current point in time. At the top of that page, there is a slider with which you can select dates and times in the recent past, as demonstrated in [Figure 15-17](#), but there is also a pull-down

menu that will allow you to choose from previous checkpoints, as well.

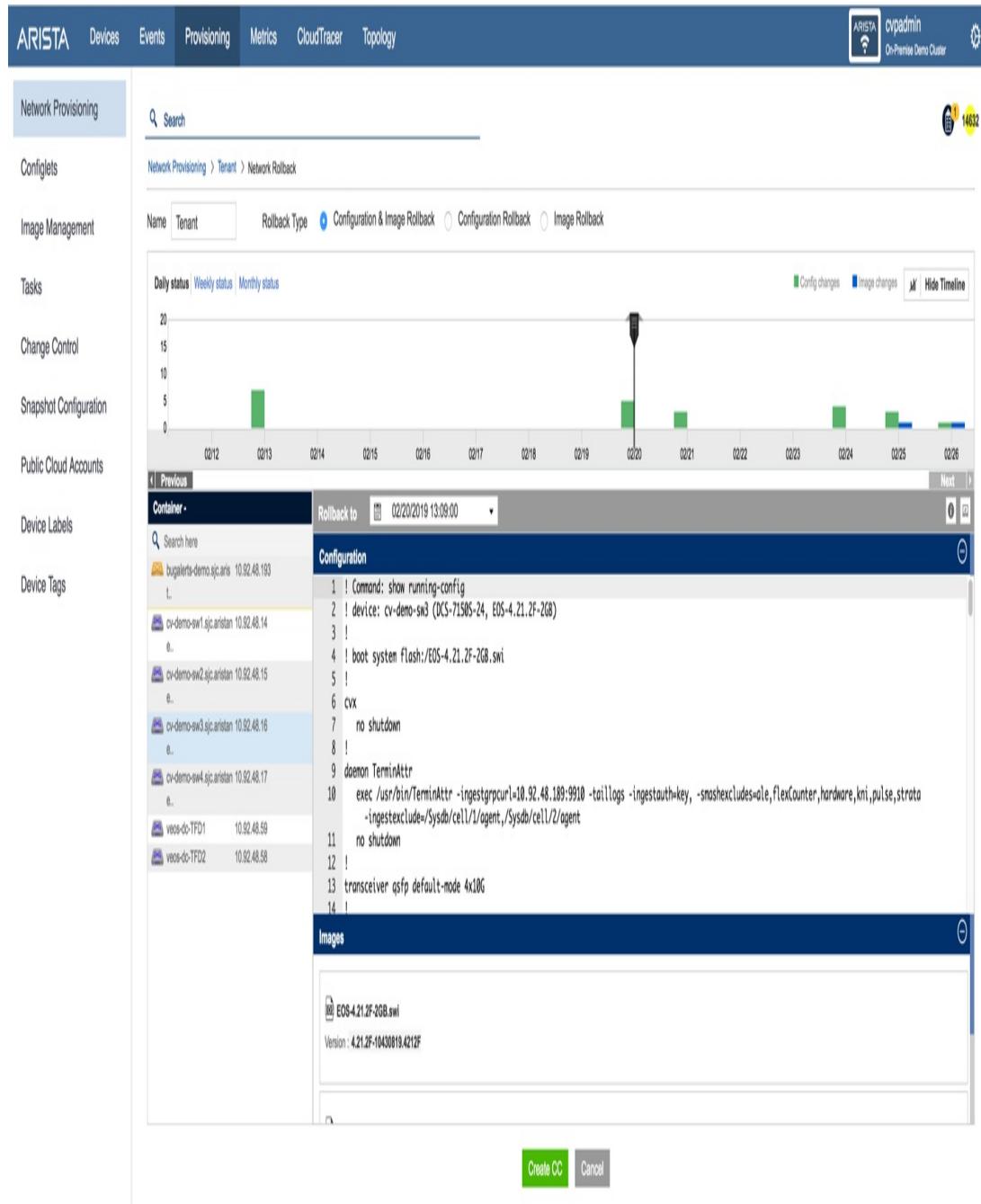


Figure 15-17. Network rollback detail

In this example, I've selected network rollback from the main tenant container. I then moved the slider on the top to the left until I arrived at

one of the green bars (February 20), which then showed me the images and configuration that will be applied. I should note that because I chose the main tenant container that there are multiple devices that will be affected. The list of devices is on the left of the pane, and the image and configuration that will be applied to the selected device is on the right.

I cannot overstate the power of this feature. With a couple of careful selections on the CVP Provisioning page (and, of course, a change control to make it happen), you can return the entire network to its state an hour, week, or month ago (or any other time period for which snapshots have been saved). Did a change-control go horribly wrong? Just roll it back. And what's particularly amazing is that you don't need to roll back the entire network. You can roll back devices, as previously covered, or any container that might or might not have other containers as children. How long would it take you to do that by hand?

Is rollback fool-proof? More important, is it GAD-proof? I don't think there's any such thing, but I can say that I've seen this used to great effect, and this feature alone would make me consider using CVP to manage configurations because it's just that good.

Don't forget that CVP isn't doing the old-fashioned *SSH to a device and pretend to be a user* way of configuring nodes. It's using eAPI and the power of *config sessions* ([Chapter 9](#)), so it works much better than other vendors' tools that you might have used from years gone by.

Events

The Events page is one of my favorite parts of CVP because it shows data in real time and the interface is terrifically easy to use and (I'll admit it) play with.

An *event* is triggered when something unexpected happens. Events are categorized with the severities Info, Warning, Error, and Critical. At the bottom of the Events page, there is a timeline, as shown in [Figure 15-18](#), that marks events and when they occurred as colored dots. The color of the event reflects its severity.

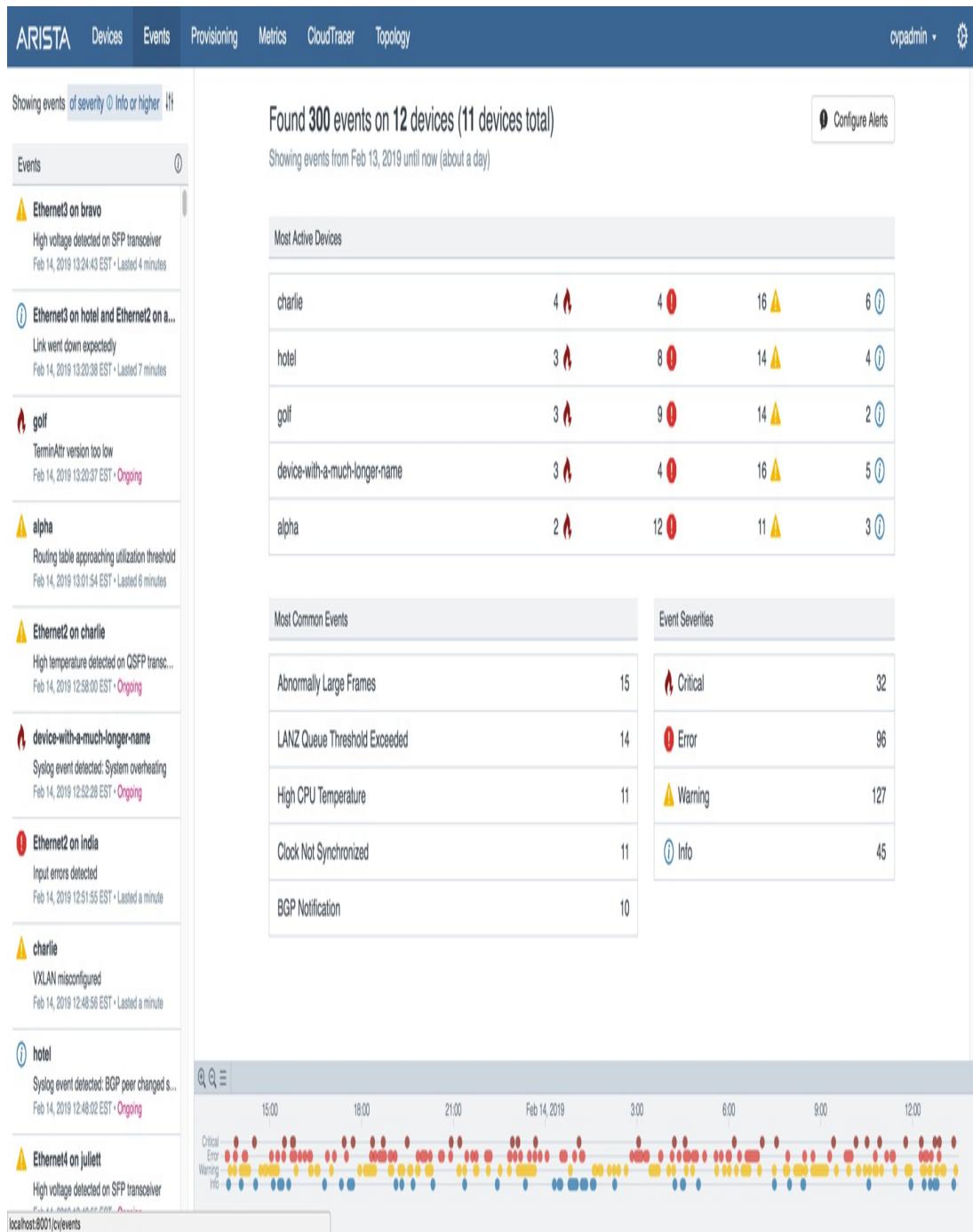


Figure 15-18. CVP events

Selecting an event on the left changes the main part of the screen to show details about that event. What I really like about this feature is the fact that CVP shows you the important part of the data for each alert type. For example, in Figure 15-19, I've selected an event for

which output errors were detected, and you can see all of the relevant graphs for that type of data. By the way, this error condition was occurring as I watched, so those pretty graphs were updating in real time. Real time! Did I mention that this is all in real time? That still blows me away.

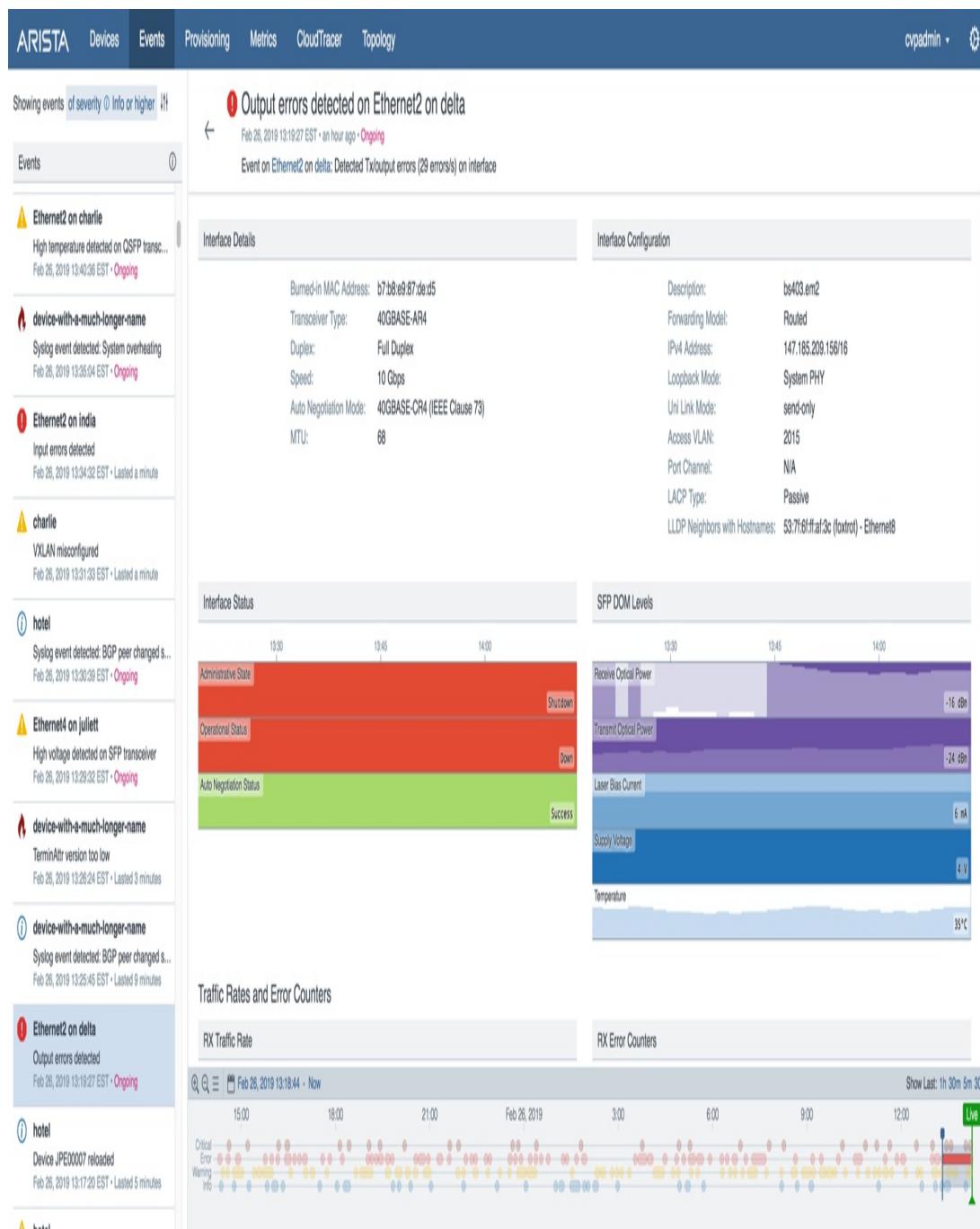


Figure 15-19. CVP event—output errors

On the bottom of the screen, that timeline has a couple of slideable bars on it that default to the time of the event. By sliding those bars, you can expand the graphs (again, in real time) to show information for a longer period so that you can get a better feel for trends as needed. There is a third bar in the middle of the other two that you can slide, which puts an indicator line on all of the graphs to show the detailed information for the data collected at that instant in time. Talk about visibility!

Moving on to another event type, [Figure 15-20](#) shows a high-CPU event. Of course, if the CPU is high, we care about different data, so this event defaults to showing us things like CPU utilization for each core along with different load averages and memory utilization.

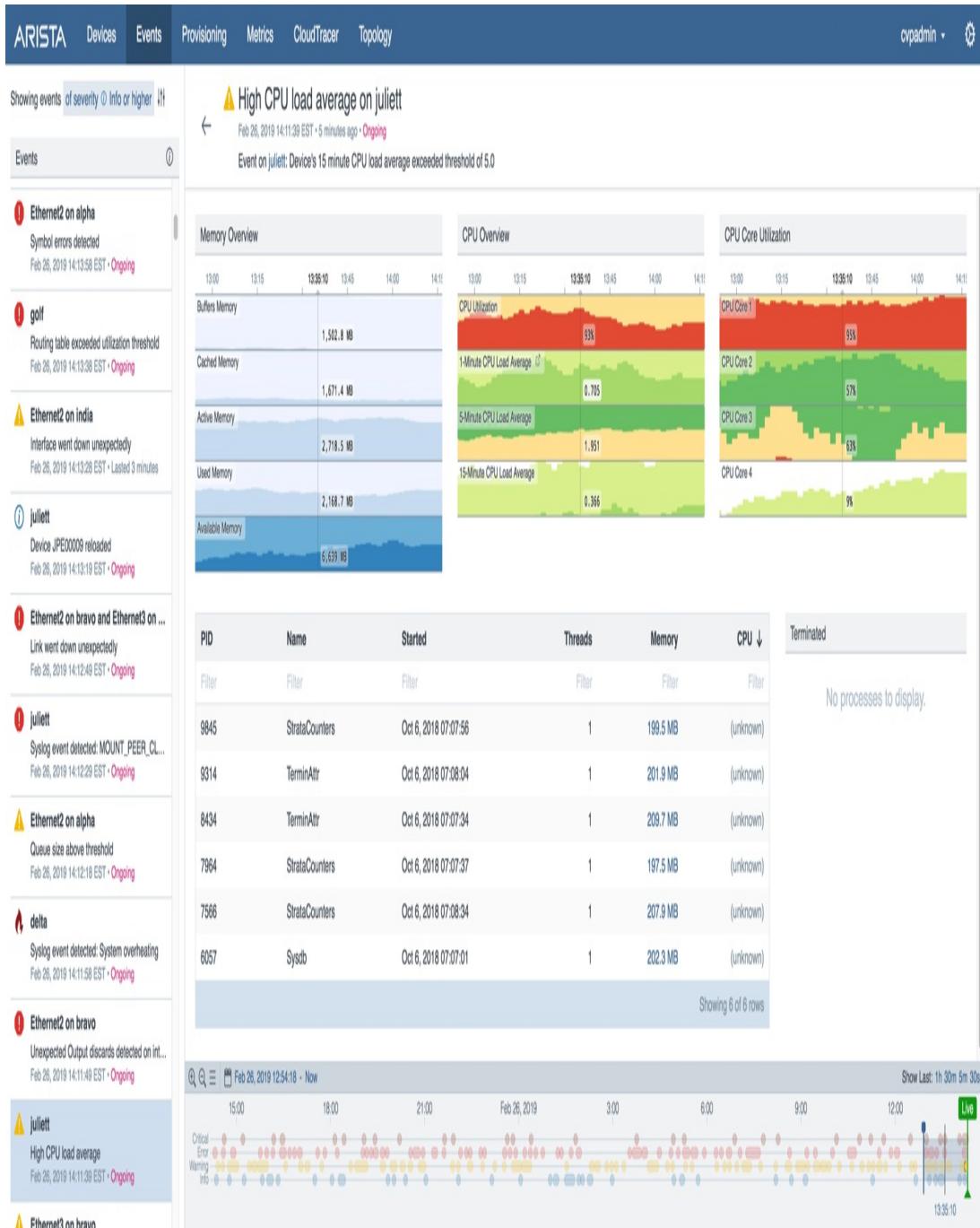


Figure 15-20. CVP event—high CPU

I know I've written this repeatedly, but as someone who has spent a lot of time in NOCs working with old-school network management tools, this is a game changer. The ability to see current information in real time along with the crazy level of historical detail that's available

makes all the other tools out there pretty darn obsolete. I think this is like some of the CLI tools available in EOS in that the more you use them, the more you'll find yourself utterly frustrated when using other vendors' tools.

Cloud Tracer

Cloud Tracer, shown in [Figure 15-21](#), presents a colorful matrix of latency information between the devices under CloudVision's purview and some devices in various cloud service providers. This single snapshot into the overall delay of the network from points in the local network to remote points in the cloud (or clouds, in this case) can give a great way to know whether complaints like "the network is slow" are a result of local or remote issues.



Figure 15-21. Cloud Tracer

As you can see on the left of [Figure 15-21](#), the current chosen metric is Latency, but HTTP Response Time, Jitter, and Packet Loss are also selectable options.

In the ever-expanding work of cloud-attached networks, this can be a terrific tool for not only diagnosis but also simple status.

Topology

Topology is the section I find myself most drawn to as someone who spent far too long using tools like HP OpenView. That's because it shows a network layout in a way that makes sense to me from years of looking at similar screens.

This view is probably not correct by default, and you might need to massage it, instructing CVP what type of *node type* each entry is, along with its *Datacenter*, *Pod*, *Flock*, and *Rack*, in order to make the output look pretty. When you've done this, though, the output is terrifically useful, so it's worth the energy to make it look the way it should. By the way, I'd never heard of the term *Flock* before, so I asked the developers who told me that it's a grouping of devices within a rack.

To give you an idea of what I mean, [Figure 15-22](#) shows a simple lab layout with all of the devices left untagged, in which case they should all remain in the Untagged *cluster*.

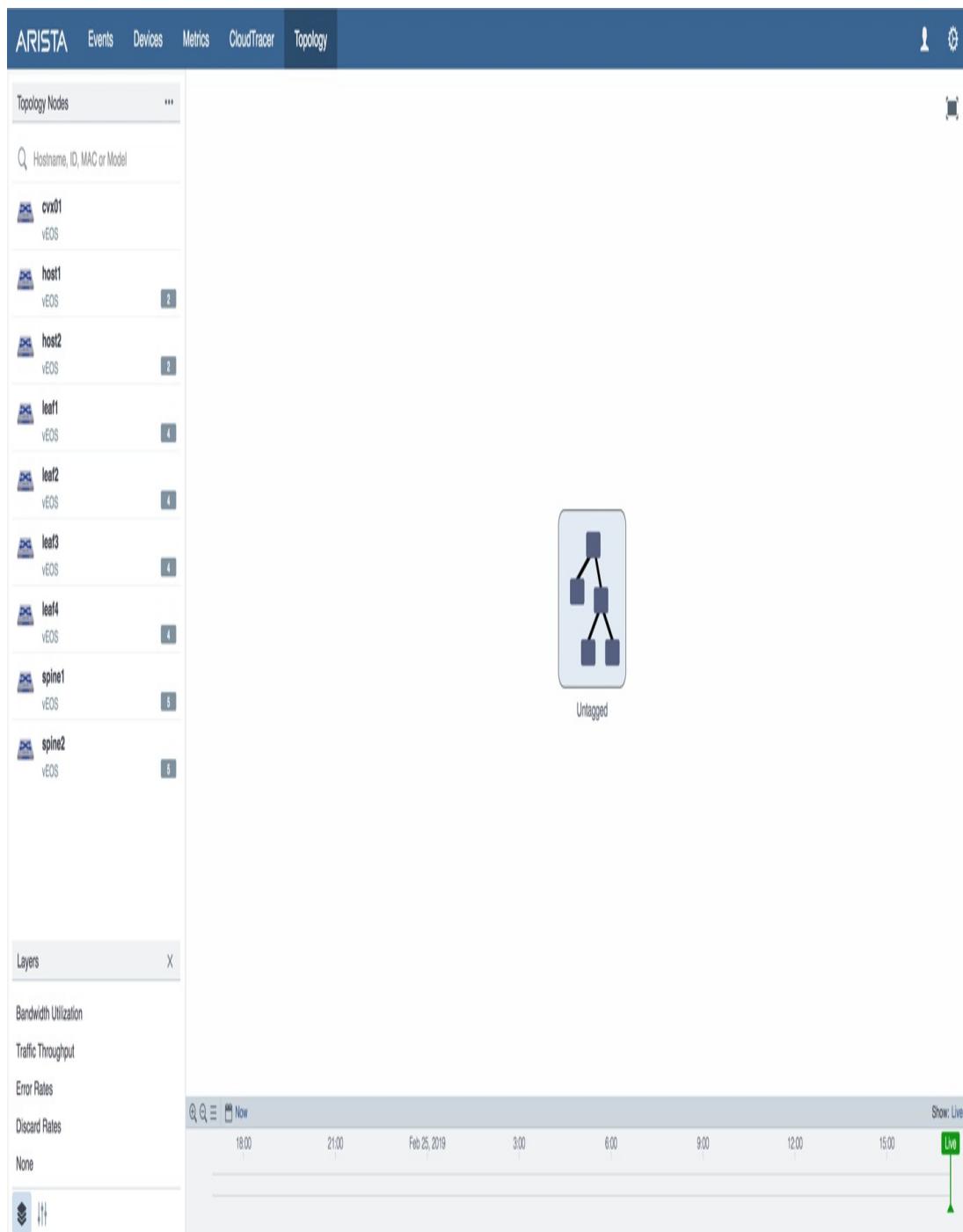


Figure 15-22. Untagged cluster in network topology

A cluster is a collection of *nodes*. Double-click a cluster to see its nodes. After double-clicking the Untagged cluster, you see the results shown in Figure 15-23.

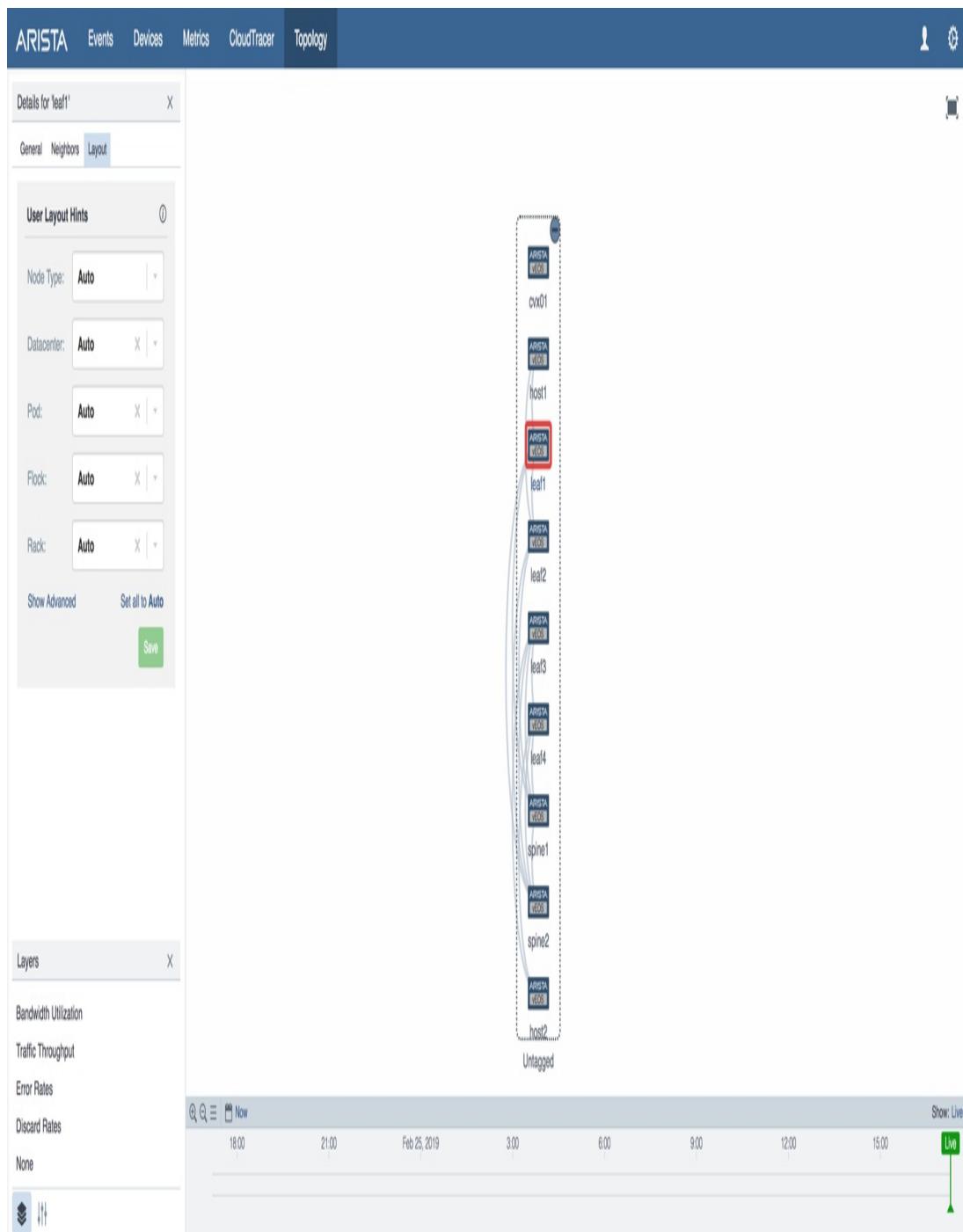


Figure 15-23. Untagged cluster expanded

Note that in Figure 15-23 I was holding the mouse pointer over the cluster, which highlights the nodes within and gives the option for collapsing it via the little minus sign in the upper right. I have also selected the leaf1 node, which is why it's highlighted.

At this point I'm going to use that dialog on the left and set each of the devices to their proper purpose such as spine, leaf, edge, core, management, or endpoint device. Between doing this and the connections learned via LLDP, the topology will begin to form. In fact, as you can see in [Figure 15-24](#), CVP has assumed some things based on the network connectivity of the leaf switches and has put them into their own rack clusters. All I did was apply the layout of *Spine* to the spine switches and *Leaf* to the leaf switches—CVP did the rest. Don't worry, you can override those labels.

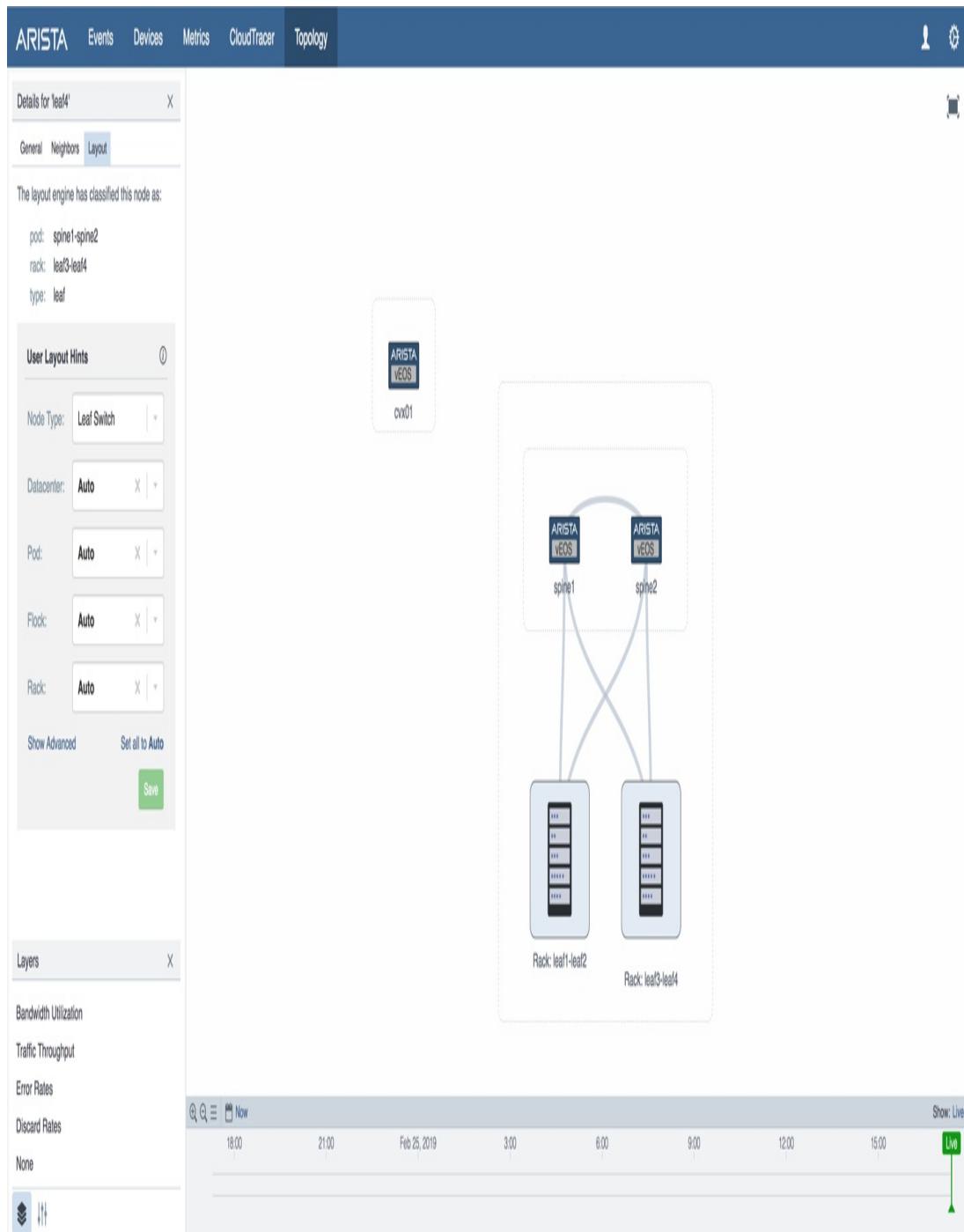


Figure 15-24. Spine and leafs put into racks automatically

As a matter of fact, CVP has gone a step further. If you look back, you'll see that there were Host1 and Host2 nodes that seem to have disappeared. CVP has figured out that they were attached to leaf switches and put them into the proper racks for us based on the rack

configured on those leafs (leaves?). [Figure 15-25](#) illustrates that the highlighted node, Host1, has been placed into the Rack1 cluster even though the layout on the left of the screen shows that the node is set to auto for all settings. Why wasn't the CVX node placed anywhere? Because it's not directly connected to any of the other devices.

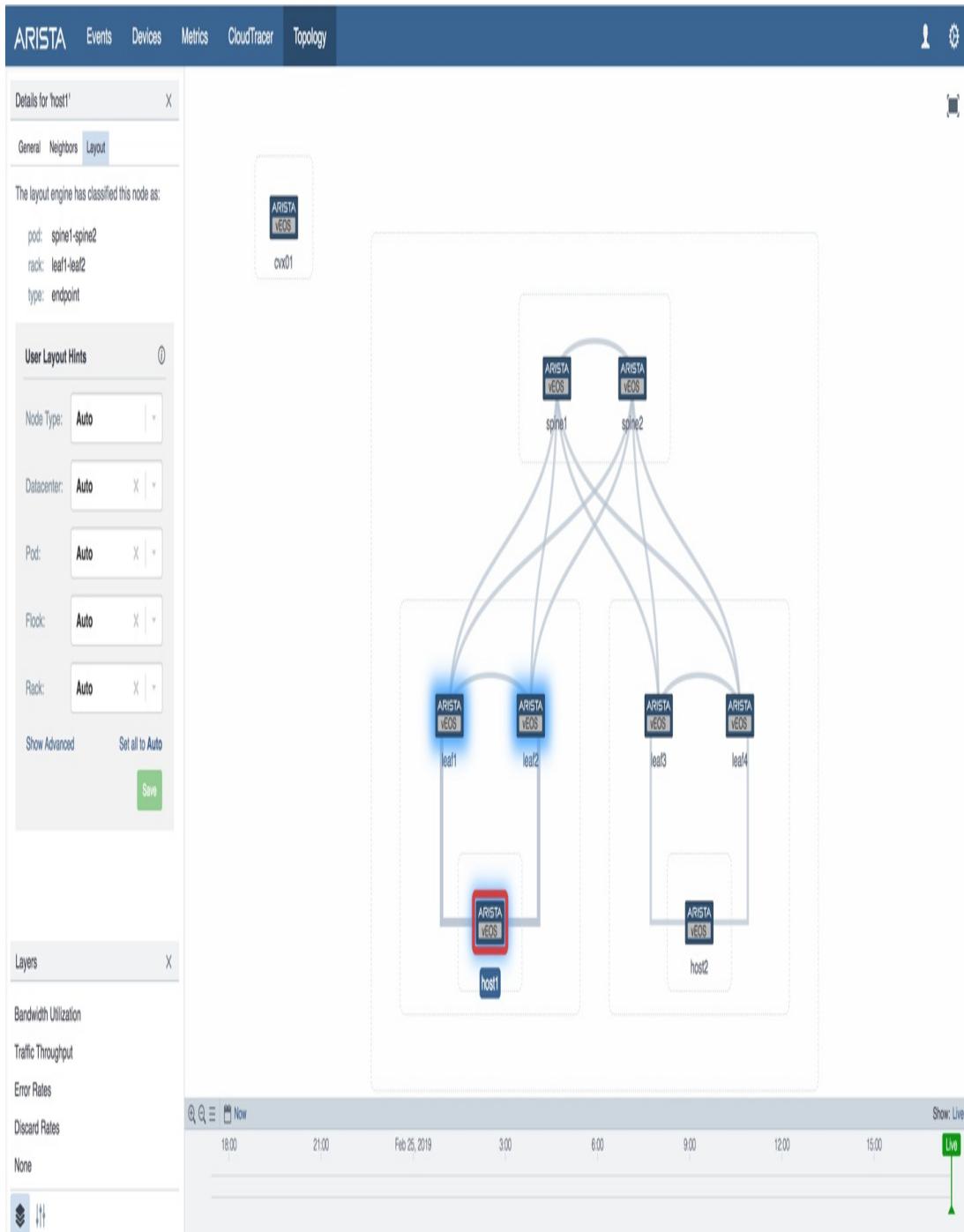


Figure 15-25. CVP places hosts into racks automatically

That floating CVX node actually brings me to one of my frustrations with Topology view, which is that the absolutely obsessive part of my brain doesn't like it there and wants to move it, but if you click it with the mouse and move it, the entire topology moves. In fact, if you're

used to working with tools like OpenView with which you can place any object anywhere, you'll need to get used to CVP placing nodes and clusters where it thinks they belong. That might change, but as of this writing I just need to learn to adapt, and as anyone who knows me will tell you, that's an uphill climb of Sisyphean scale. Even sitting here looking at the image, I want to move that node.

How big can these topologies be? Pretty darn big. [Figure 15-26](#) shows a fairly large network, but what's really more interesting about that figure is that the internode links are highlighted. That can be difficult to discern in a black-and-white book (here's where you soft-copy buyers get your money's worth!), but the majority of the links in [Figure 15-26](#) are colored green, whereas a couple near the bottom are highlighted in red. These colors are providing a quick glance of bandwidth utilization.



Figure 15-26. Topology view with bandwidth utilization

The colored links can show other information, as well. Figure 15-27 shows a CVP Topology with links in red indicating network errors that have been discovered. Remember, CVP doesn't use SNMP to gather data, so this information was learned and displayed in real time.

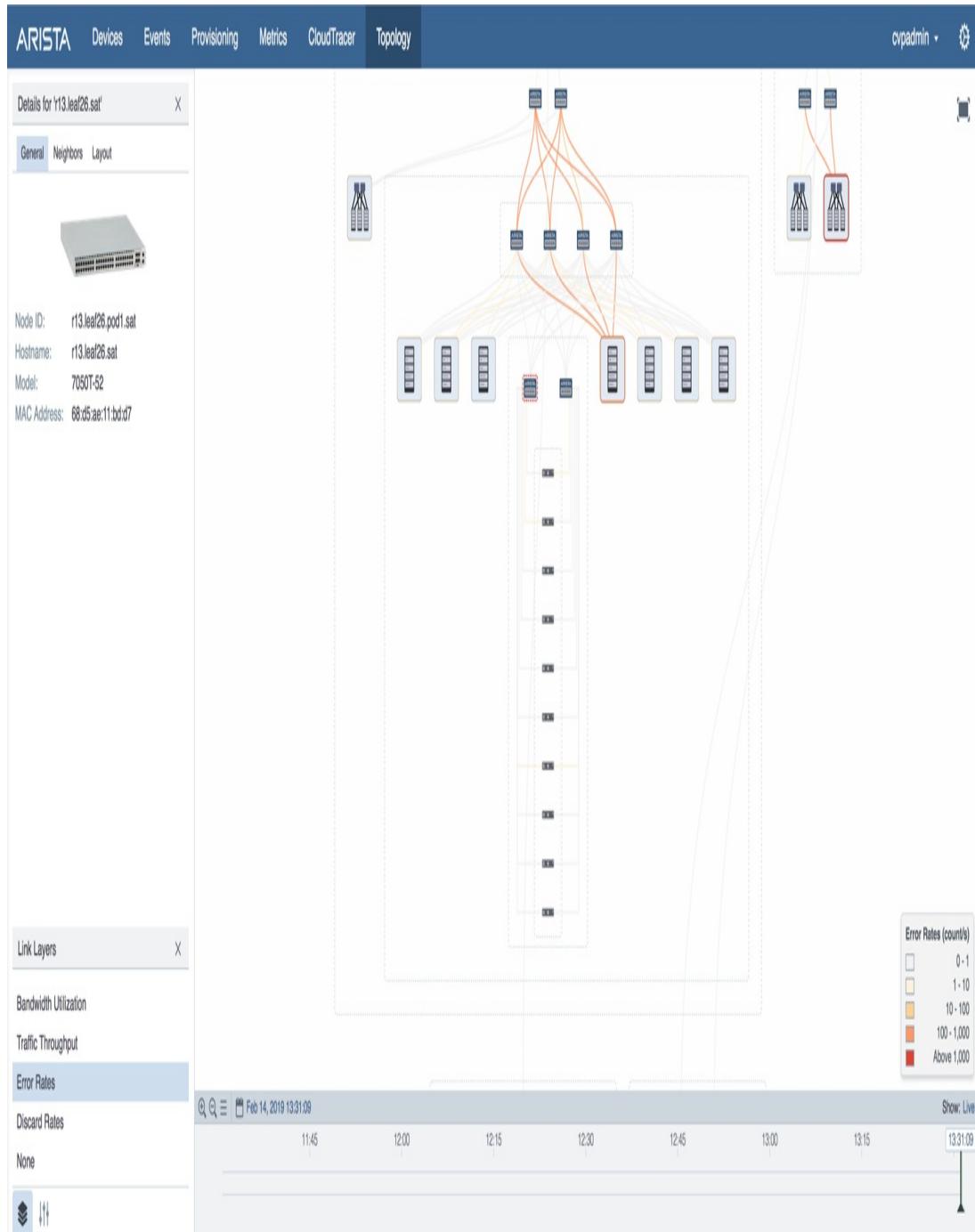


Figure 15-27. Topology view zoomed in with errors highlighted

Back to how large the topology can be, Figure 15-28 shows an even larger network. Take a good look at that image, though, and see how many of those objects are actually clusters that have more nodes inside them.



Figure 15-28. A large CloudVision deployment

When it comes to number of devices that CVP can support, those numbers are changing too often for me to publish here, but I will say that every time the numbers do change, they change in a positive

direction, and usually the published scale numbers double, so if CVP doesn't scale to fit your needs today, please talk to your sales team, because it likely will in the near future.

Conclusion

CloudVision is a next-generation tool that is absolutely not a single pane of glass, because that would be a terrible thing to call this tool that gives you so much visibility into a single...well, window.

As someone who is a die-hard CLI-mangler, the Network Provisioning side of it doesn't interest me as much as the Network Topology and Events parts do, but even if I'm not necessarily drawn to the provisioning aspect, I must say that I do like the Tasks and Change Control features, and that could be enough to get me to convert, especially in a larger-scale enterprise environment.

As someone who prefers to write his own automation scripts, I can manage my networks without Network Provisioning, but that's for me and my labs. When it comes to a larger network, and to me this tool really shines in the enterprise space, CloudVision is a gem.

What really excites me, if you haven't gotten the message yet, is that all of the Events and Telemetry data is streamed in real time. Add to that the real-time ability to see bandwidth utilization and error indicators in the Topology view and there's absolutely something for everyone in CloudVision.

Chapter 16. The EOS Extension System

EOS is the Extensible Operating System, so let's have some fun and extend it!

OK, so my idea of fun is installing extensions into a networking switch. What can I say? I'm living the dream.

Extensions are nothing more than one or more RPMs zipped up with a manifest file. These extensions have the filename extension `.swix`, for SoftWare Image eXtension. For those who are not familiar with Linux, RPM stands for *RPM Package Manager*. If that makes you twitch because your fifth-grade teacher wouldn't let you use the word being defined within the definition, you're not alone. RPM originally stood for *Red Hat Package Manager*, which is far less likely to offend a fifth-grade teacher's sensibilities. The reason for the change is that RPMs are used in many more operating systems these days, even if the recursive name gives me hives.

RPMs are *packages* that usually contain compiled code. Using an RPM is just like downloading a program from the internet that needs to be unpacked and installed. On a Windows machine, you might download an Installer, whereas on a Mac, you might download a DMG image that contains a PKG package file. The idea is the same. RPMs are just the way Linux handles packages. Because EOS is running on Linux,

it's only natural to use RPMs to add extensions. If you're familiar with Linux, you probably know about repositories that allow you to do things like add packages and all of the required dependencies with ease. Arista disables access to these repositories for the very valid reason that doing something like upgrading Linux might break EOS. To counter this, the extension system exists, which allows you to install packages with all of their dependencies in a single file. You can also use the extension system to install single RPMs, which is how Arista deploys security patches.

So, what sort of extensions can you add? If you are a skilled programmer, I'd say that you can add anything you want! In fact, we'll be installing a package that I wrote. A quick place to look for existing extensions is on Arista's EOS Extensions web page, which you can find on the [EOS Central site](#). You also can search GitHub for Arista extensions or just go to the [software downloads page](#) on [arista.com](#). Any extensions I write are on my [GADify GitHub page](#).

For this example, I use the extension `CPU-Hist`, which I wrote. This extension adds functionality similar to Cisco's `show proc cpu hist` command using `gnuplot` (another extension that we need to add).

First, I need to download the package. I have a lab server set up, so I just used that to get my files.

With my hostname set and the file located on my server, I can copy directly from the web server to my switch. To accomplish this, I do what most of us do and look at what my options are by using the `?` character at the command prompt:

```

Arista#copy ?
  boot-extensions      Copy boot extensions configuration
  certificate:         Source file path
  clean-config         Copy from clean, default, configuration
  drive:               Source file path
  extension:           Source file path
  file:                Source file path
  flash:               Source file path
  ftp:                 Source file path
  http:                Source file path
  https:               Source file path
  installed-extensions Copy installed extensions status
  running-config       Copy from current system configuration
  scp:                 Source file path
  sftp:                Source file path
  startup-config       Copy from startup configuration
  system:              Source file path
  terminal:            Source file path
  tftp:                Source file path

```

As you can see, there are a lot of options. For this example, I use `http:`, but note that you can copy from secure websites (`https:`), secure copy (`scp:`), and a bunch of other cool options. I include the entire URL and then include the destination `extension:`

```

Arista#copy http://10.0.0.100/RPM/CPU-Hist-1-4.swix extension:
Copy completed successfully.

```

If the file is small, as this one is, you might see a message flash so quickly that you can't see it. Don't worry too much about it; it's more important for huge files or for files being transferred over slow connections. If you're really wondering (as I was) what it says, here's a snapshot while downloading a generic huge file:

```

Arista#copy http://www.gad.net/RPM/HugeFile.zip extension:
'RPM/HugeFile.zip' at 5235283 (3%) [Receiving data]

```

Because this extension actually requires another extension, I need to

grab that one, as well:

```
Arista#copy http://10.0.0.100/RPM/GnuPlot-EOS-4-15-2.swix
extension:
Copy completed successfully.
```

NOTE

I should point out that I could add `GnuPlot` to my own `CPU-Hist` extension and make it a single installation, but the way I wrote it allows for it to work in text-only mode for sites that might not want to install `GnuPlot`.

To see what's in your `extension:` drive, you can use the `dir extension:` command from EOS:

```
Arista#dir extension:
Directory of extension:/

-rwx        13895      Apr 17 21:10  CPU-Hist-1-4.swix
-rwx      4375995      Apr 17 21:10  GnuPlot-EOS-4-15-2.swix

3564371968 bytes total (2246819840 bytes free)
```

A much better command to use is the `show extensions` command:

```
Arista#sho extensions
Name          Version/Release  Status extension
-----
CPU-Hist-1-4.swix      1/4          A, NI    1
GnuPlot-EOS-4-15-2.swix 1.10.0/1.fc14  A, NI    18

A: available | NA: not available | I: installed | NI: not installed |
F: forced
```

This command shows some great information about the extensions, including the version, if it's available, and whether it's installed.

NOTE

If you're using a chassis-based switch like the Arista 7500s, you'll need to copy extensions on all supervisors.

So now that we have our RPMs, let's get those suckers installed. To do so, simply use the `extension` command, followed by the name of the extension. You can use the `?` character to get a list of available extensions:

```
Arista#extension GnuPlot-EOS-4-15-2.swix
Arista#extension CPU-Hist-1-4.swix
```

Success! If your extension has modified anything at the EOS command line, you'll need to log out and back in to see the effects, though extensions that do this generally warn you about this fact.

With our extensions successfully installed, let's check the status again by using the `show extensions` command:

```
Arista#show extensions
-----
```

Name	Version/Release	Status	extension
CPU-Hist-1-4.swix	1/4	A, I	1
GnuPlot-EOS-4-15-2.swix	1.10.0/1.fc14	A, I	18

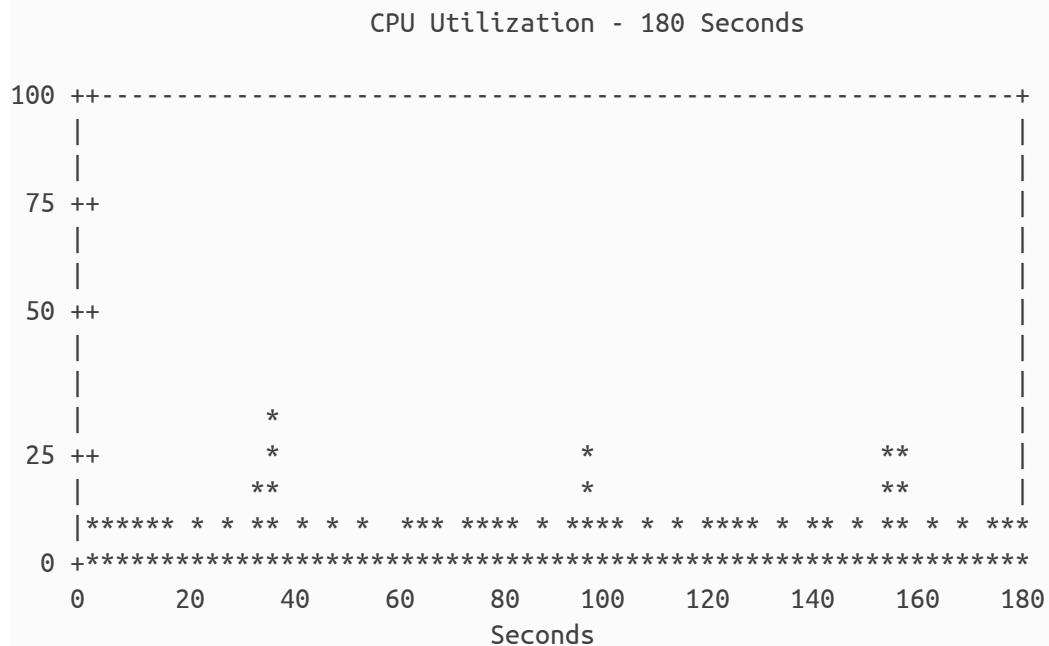
```
A: available | NA: not available | I: installed | NI: not installed |
F: forced
```

The flag under the `Status` column now shows an `I` (installed), whereas before we installed it, it showed an `NI` (not installed).

So, what do these extensions get us, anyway? The `GnuPlot` extension

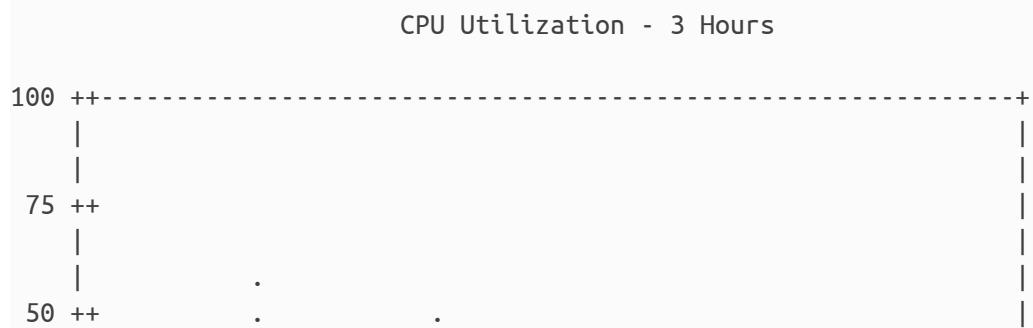
basically installed the GnuPlot open source graphing tool and all of its dependencies onto the switch. The **CPU-Hist** extension records CPU utilization and uses GnuPlot to graph it. The **CPU-Hist** tool accesses with the **cpuhist** command, which is actually an alias installed by the RPM during the extension installation process:

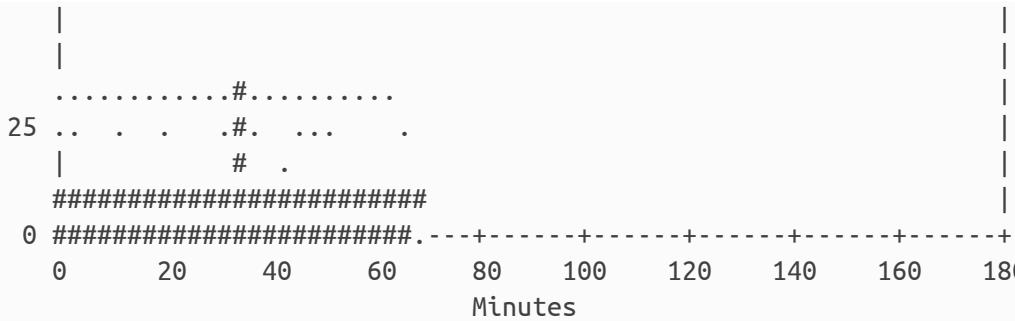
```
Arista#cpuhist sec
```



Here's the output of **cpuhist min** after about an hour:

```
Arista#cpuhist min
```





To remove an extension, just negate the extension command used to install it:

```
Arista#no extension CPU-Hist-1-4.swix
Arista#no extension GnuPlot-EOS-4-15-2.swix
```

Note that this does not remove the extension; instead, it just makes them inactive:

Arista#sho extensions			
Name	Version/Release	Status	extension
CPU-Hist-1-4.swix	1/4	A, NI	1
GnuPlot-EOS-4-15-2.swix	1.10.0/1.fc14	A, NI	18

A: available | NA: not available | I: installed | NI: not installed | F: forced

If you want to completely remove an extension, delete it from the *extensions*: device:

```
Arista#del extension:CPU-Hist-1-4.swix
Arista#del extension:GnuPlot-EOS-4-15-2.swix
Arista#sho extensions
No extensions are available
```

If you're wondering what the *extensions*: device really is, it is nothing more than a hidden directory on *flash*:

```
Arista#bash ls -al /mnt/flash | grep extensions
drwxrwx--- 2 root eosadmin 4096 Nov 27 21:49 .extensions
```

There is a limitation of extensions: their status of being installed does not survive a reboot. If you'd like extensions to be installed when the system boots, use the `copy installed-extensions boot-extensions` command. Here I've put them both back, installed them, and then configured them to load after boot:

```
Arista#copy http://10.0.0.100/RPM/GnuPlot-EOS-4-15-2.swix
extension:
Copy completed successfully.
Arista#copy http://10.0.0.100/RPM/CPU-Hist-1-4.swix extension:
Copy completed successfully.
Arista#extension CPU-Hist-1-4.swix
Arista#extension GnuPlot-EOS-4-15-2-min.swix
Arista#copy installed-extensions boot-extensions
Copy completed successfully.
```

“Boot extensions” is really just a file on *flash* that contains a list of extensions to be loaded:

```
Arista#dir boot*
Directory of flash:/boot*
-rwx 27 Nov 22 17:00 boot-config
-rwx 46 Nov 27 21:53 boot-extensions

3564371968 bytes total (2246811648 bytes free)
```

Here's what the `boot-extensions` file looks like:

```
Arista#more flash:boot-extensions
CPU-Hist-1-4.swix
GnuPlot-EOS-4-15-2-min.swix
```

Even on EOS version 4.21.1F, there is seemingly no way to get rid of the boot extension entry for a deleted extension. There is no `del`

`boot-extensions` command available, and I could find no other way to delete it from within the command-line interface (CLI). Luckily, this is EOS, and there's always another way. The list of items in the `boot-extension` directory is simply a list of extension names found in `flash`. Here's the contents of the boot extensions file from Bash:

```
Arista(config)#bash  
Arista Networks EOS shell  
[admin@Arista ~]$ more /mnt/flash/boot-extensions  
CPU-Hist-1-4.swix  
GnuPlot-EOS-4-15-2-min.swix
```

To delete one line, use vi from Bash and delete the line. In the past, if you just deleted the file, you'd get an error when trying to view it, but that is no longer the case in modern EOS:

```
[admin@Arista ~]$ cd /mnt/flash  
[admin@Arista flash]$ mv boot-extensions GAD  
[admin@Arista flash]$ exit  
logout  
Arista(config)#  
Arista(config)#  
Arista(config)#sho boot-extensions  
Arista(config)#
```

There are many other cool extensions out there, from Apache to DHCP, and myriad other cool tools. You can find many of them in the [EOS Central repository](#) (registration required). Even if you don't extend your switch using something like my `CPU-Hist` package, if Arista releases a patch for a bug fix or security advisory, it will usually be installed using the extension system.

Here's a snapshot of one of my switches running an older version of

code that has not only CPU-Hist installed, but also the Apache server as well as a patch for Arista Security Advisory #15 (you can [view security advisories on arista.com](#)):

```
Old-Arista#sho extensions
```

Name	Version/Release	Status	RPMs
CPU-Hist-1-3.swix	1/3	A, I	1
GnuPlot-4.4.0.swix	1.2.14/11.fc14	A, I	42
httpd.swix	2.2.17/1.fc14	A, NI	7
secAdvisory0015.swix	1.0.0/SA15	A, I	1

```
A: available | NA: not available | I: installed | NI: not installed F: forced
```

One final note about extensions is in order. Extensions are loaded very early in the boot process, which means that you can actually have CLI commands in the *startup-config* that rely on them.

Conclusion

When I first started at Arista, I envisioned a world in which everyone was writing software extensions that could be used to make EOS even better. Although that does happen, most of the time I see the extension system being used for things like security or bug patches. The big players, though are absolutely writing custom extensions for their switches, and even their own agents, through something called the EOS SDK, but that is outside the scope of this book. Talk to your account team if you'd like to know more about the EOS SDK.

Chapter 17. Multiple Spanning Tree Protocol

Spanning Tree Protocol (STP) is very important in Layer 2 (L2) networks, and its impact should be clearly understood when designing or even troubleshooting a network. If you've been around Cisco gear for the majority of your networking life, you've probably used Per-VLAN Spanning Tree (PVST) or Rapid-PVST (RPVST). In this chapter, I cover a form of STP that is becoming more common in large data centers: Multiple Spanning Tree (MST).

Data center networks have very different requirements than those of enterprise networks. I worked for a client that had Cisco 3750s in the core of a small data center. Things seemed to work great until the client added the 257th VLAN, and that's when they learned that Cisco 3750s support STP only up to 256 VLANs. Bummer. I should note that this is not a knock on the 3750, but rather the idea that someone decided to use a device designed to be an office switch as a data center core.

I was brought in to help solve the problem, and after my recommendation of, "Buy data center-class chassis switches" was ignored, I looked for other options. That's when I learned about MST.

NOTE

You might see both MST and MSTP (Multiple Spanning Tree Protocol) referenced. I tend to prefer MST because it's easier to say, but EOS uses MSTP likely because they're

commands that affect how the protocol is used.

Arista switches can run a variety of STP types, but they default to MST. You can change the type by using the **spanning-tree mode mode-type** command:

```
SW1(config)#spanning-tree mode ?
  backup      Backup port mode
  mstp       Multiple spanning tree protocol
  none       Disable spanning tree
  rapid-pvst  Per VLAN rapid spanning tree protocol
  rstp       Rapid spanning tree protocol
```

Because I've covered Spanning Tree and rapid-PVST in *Network Warrior*, I'm going to focus on MST in this chapter. Buckle up, because if you've never used MST, you're in for a fun ride.

MST

MST is a version of STP that is much simpler than PVST, yet I rarely see it implemented. I think a big part of that is due to the fact that there is very little documentation out there, and what is available, well, let's just say that reading ancient Sumerian would be easier.

MST can appear to be very difficult, but it doesn't need to be. Like anything technical, if you make it too complicated, it will be difficult to understand, so keep it simple and you'll do fine. MST can be simpler to configure, easier to manage, often makes more sense (when deployed simply), and uses fewer CPU cycles than PVST. When I discovered that Arista switches were configured with MST as the default, it was

love at first sight.

Hey, I never said I was normal.

MST is based on Rapid Spanning Tree Protocol (RSTP), which is a great thing because it offers some great backward compatibility, as you'll see later in this chapter. You might be thinking, "Well, isn't PVST multiple Spanning Trees?" And you'd be right, but MST works differently. The problem with PVST is that there's a Spanning Tree instance running for every VLAN. On an enterprise network comprising 20 or 30 (or even 100) VLANs, that's not a big deal, but in a data center where there can easily be hundreds of VLANs, things can become problematic.

First, each Spanning Tree instance requires CPU and memory resources, and unless you're a fan of the old *even VLANs on the left and odd VLANs on the right* paradigm, all of the VLANs will likely have the same topology anyway. Suppose that you are a fan of splitting your VLANs up between two core switches (I am not); then you have two topologies (assuming that all trunks allow all VLANs, of course). Why have potentially hundreds of Spanning Tree instances running when there are only two topologies?

NOTE

I have never been a fan of splitting up STP topologies with odd/even VLANs. It makes troubleshooting more difficult and simply isn't needed in most modern networks. Fifteen years ago, when switches were slow and STP shut down half of the available links, I could see needing to distribute the load, but in a modern network, I'll take simplicity every time. Besides, with Multi-Chassis Link Aggregation Group (MLAG), Virtual Address Resolution Protocol (VARP), and other cool features available in Arista switches, there's no need for

any of this manual load-balancing nonsense anymore.

MST allows you to have one Spanning Tree instance for all your VLANs. Or, if you’re still hell-bent on balancing which switch is the root based on some arbitrary pattern such as even/odd, you can split up your VLANs into *instances*, each with its own root. Oh, and MST is a standards-based protocol. There are no proprietary things to worry about, and it works with any vendor’s switch that supports it.

NOTE

There are some vendor-interaction complexities with MST, as you’ll see later in this chapter, but rest assured, MST is still an open standard.

The MST documentation goes into painful detail about MST, the Common Spanning Tree (CST), the Internal Spanning Tree (IST), the MSTI, the Common Internal Spanning Tree (CIST), and who knows what else. I explain all this to you later in this chapter and show you why it doesn’t need to be all that difficult. Naturally, I’ll do this in a way that makes sense to me and seems to make sense to people I teach. My goal is not to get you certified; my goal is to make you understand.

Let’s begin with a simple scenario to see how MST works. In Figure 17-1, there are two switches: SW1 and SW2. These two switches are connected together with two links.

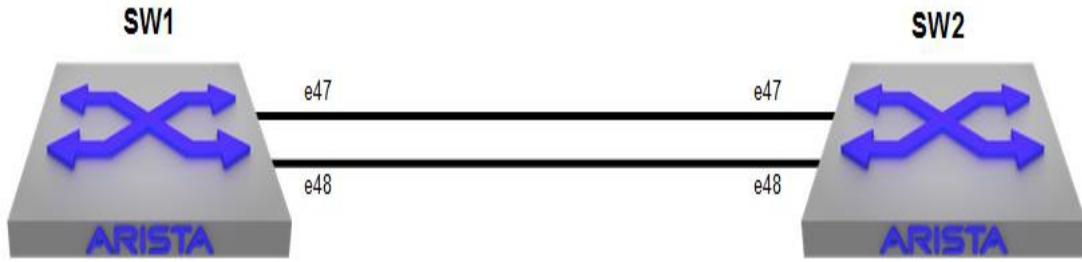


Figure 17-1. Two switches connected with two links

Of course, such a topology is a problem because it creates a loop, and loops are bad. Spanning Tree, in its default state on Arista switches, shuts down one of the links, as shown in Figure 17-2. For details about which link is blocked, I recommend the excellent chapter on Spanning Tree in *Network Warrior* (can that guy write or what?). For now, suffice to say that the blocked port is the interface farthest from the root bridge. In my little network, SW1 won the battle for root bridge status (everything is set to defaults for now), so the e48 interface on SW2 ended up blocking.

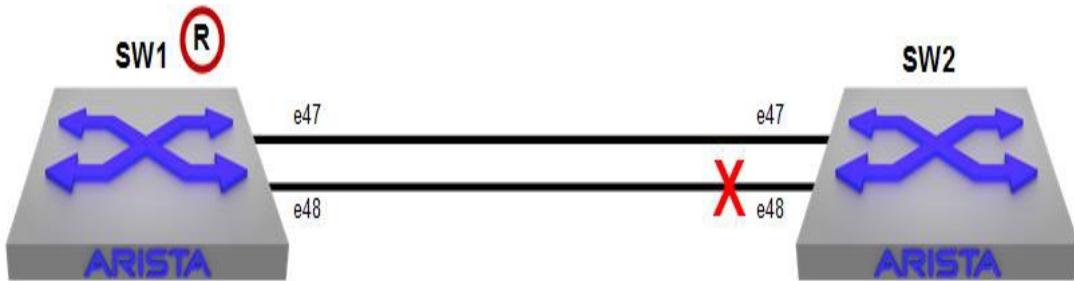


Figure 17-2. STP saving the day by breaking the loop

This simple network shouldn't be a surprise to anyone who has ever used Spanning Tree, but the difference is that out of the box, Arista switches use MST, so let's take a look at what the switches report.

The command to show STP information is `show spanning-tree`. Here's the output for SW1:

```

SW1#sho spanning-tree
MST0
  Spanning tree enabled protocol mstp
  Root ID  Priority  32768
            Address  2899.3a18.46f1
            This bridge is the root

  Bridge ID  Priority  32768 (priority 32768 sys-id-ext 0)
            Address  2899.3a18.46f1
            Hello Time 2.000 sec  Max Age 20 sec  Forward Delay 15 sec

  Interface      Role      State      Cost      Prio.Nbr  Type
  -----+-----+-----+-----+-----+-----+
  Et47          designated  forwarding  2000      128.47    P2p
  Et48          designated  forwarding  2000      128.48    P2p

```

This output shows the information we'd need when looking at STP, including the root bridge's information, which includes the root bridge's priority (the default is 32,768), the root bridge's MAC address, this bridge's priority, this switch's MAC address, and the status of every interface that has received Bridge Protocol Data Units (BPDUs). This switch shows that it is the root and the bridge information for itself. The ports on this switch are both designated because it's the root bridge.

Here's the output for SW2:

```

SW2#sho spanning-tree
MST0
  Spanning tree enabled protocol mstp
  Root ID  Priority  32768
            Address  2899.3a18.46f1
            Cost      0 (Ext) 2000 (Int)
            Port      47 (Ethernet47)
            Hello Time 2.000 sec  Max Age 20 sec  Forward Delay 15 sec

  Bridge ID  Priority  32768 (priority 32768 sys-id-ext 0)
            Address  2899.3a26.481d
            Hello Time 2.000 sec  Max Age 20 sec  Forward Delay 15 sec

```

Interface	Role	State	Cost	Prio.Nbr	Type
Et47	root	forwarding	2000	128.47	P2p
Et48	alternate	discarding	2000	128.48	P2p

This output clearly shows that interface Et47 is the root port and that interface Et48 is the alternate root port, which is blocking (discarding).

The thing to notice is the first line of output on both switches. On this line, they each say *MST0*.

MST0 is the default *instance* in MST. I cover instances in a minute, but the thing to remember now is that MST0 is always on, it's always active, and every interface forwards MST0 BPDUs. You cannot disable MST0. That will become important later in this chapter, but for now, understand that because MST0 is always on, if you connect two switches running MST together like I have, MST will run and behave as you'd expect.

Let's change the network so that SW2 is the root bridge by using the command that you likely already known, `spanning-tree root primary`:

```
SW2(config)#spanning-tree root primary
```

This has the almost immediate effect of making SW2 the root. Here's the proof that the STP bridge priority is now 8,192:

```
SW2(config)#sho spanning-tree
MST0
  Spanning tree enabled protocol mstp
  Root ID    Priority     8192
```

```

Address      2899.3a26.481d
This bridge is the root

Bridge ID  Priority      8192  (priority 8192 sys-id-ext 0)
Address      2899.3a26.481d
Hello Time   2.000 sec  Max Age 20 sec  Forward Delay 15 sec

Interface      Role      State      Cost      Prio.Nbr  Type
-----      -----
Et47          designated  forwarding  2000      128.47    P2p
Et48          designated  forwarding  2000      128.48    P2p

```

I say *almost immediate* because MST is *fast*. I've done mass switch migrations to MST and was amazed that after changing more than 200 switches, we never once noticed an outage or disruption of service. Like RSTP, MST has much tighter timers than traditional Spanning Tree. This speed is one of the things that I like about MST, and it is one of the reasons that I recommend its use.

Another way to change the priority is by using the **spanning-tree priority priority-value** command. Here, I goose up the priority a notch to 4,096:

```

SW2(config)#spanning-tree priority 4096
SW2(config)#sho spanning-tree
MST0
  Spanning tree enabled protocol mstp
  Root ID    Priority      4096
  Address      2899.3a26.481d
  This bridge is the root

  Bridge ID  Priority      4096  (priority 4096 sys-id-ext 0)
  Address      2899.3a26.481d
  Hello Time   2.000 sec  Max Age 20 sec  Forward Delay 15 sec

  Interface      Role      State      Cost      Prio.Nbr  Type
  -----      -----
  Et47          designated  forwarding  2000      128.47    P2p

```

Et48	designated	forwarding	2000	128.48	P2p
------	------------	------------	------	--------	-----

Note that the **global spanning-tree priority** command works only on MST0, which will make more sense in a bit.

To that end, I'd like to make SW1 the root bridge again because I can't sleep knowing that a higher numbered switch (SW2 versus SW1) is in charge. I just negate the commands, and MST will recalculate:

```
SW2(config)#no spanning-tree priority 4096
SW2(config)#no spanning-tree root primary
SW2(config)#sho spanning-tree
MST0
  Spanning tree enabled protocol mstp
  Root ID  Priority  32768
            Address  2899.3a18.46f1
            Cost     0 (Ext) 2000 (Int)
            Port     47 (Ethernet47)
            Hello Time 2.000 sec  Max Age 20 sec  Forward Delay 15 sec

  Bridge ID Priority  32768 (priority 32768 sys-id-ext 0)
            Address  2899.3a26.481d
            Hello Time 2.000 sec  Max Age 20 sec  Forward Delay 15 sec

  Interface  Role      State       Cost      Prio.Nbr Type
  ----- -----
  Et47       root      forwarding 2000      128.47   P2p
  Et48       alternate discarding 2000      128.48   P2p
```

One of the things that is not terribly obvious is that MST does not care about VLANs in its default state. Whenever I explain MST to someone, this is one of the most difficult things for them to wrap their heads around, likely because we've all used PVST for so long.

Remember when I wrote that MST0 is active on all interfaces at all times? This is because MST, in its basic form, has no concept of

VLANs. So long as MST0 can see another bridge (or BPDUs from that bridge, to be precise), there's a link to that bridge, and MST will act accordingly.

One of the other things special about MST0 is that it will interact with a switch that's running RPVST. How can it do this if it doesn't comprehend VLANs? That depends on the vendor, but in a nutshell, the switches will interoperate because MST is based on RSTP.

On a Cisco switch, MST0 sends identical BPDUs out every VLAN on every interface. How is that not RPVST? The difference is that it sends the *same* BPDU out every VLAN (that of MST0), not VLAN-specific BPDUs. This is a very important distinction to understand because it can bite you if you're not careful. With the Cisco model, if you configure MST0 with a priority of 4,096, it will likely become the root bridge on every VLAN on an attached PVST switch.

NOTE

I worked in a network that had STP problems due to old switches that didn't support more than 256 STP-active VLANs. To counter this, the switches turned STP off on new VLANs, which was suboptimal to say the least. When we migrated to MST, the core (which should be migrated first) went from advertising BPDUs on only 256 VLANs to advertising BPDUs on all 400 VLANs! The CPU on the core went down because it didn't need to process 400 different BPDUs on 400 VLANs, but the attached switches' CPUs went through the roof because they suddenly had 400 VLANs worth of BPDUs to deal with. The attached switches, which were still running PVST, had no idea about MST yet, so they had to process almost twice as many BPDUs as before, and because STP was a CPU-based process on those devices, their CPU utilization skyrocketed. After we moved them to MST, they settled down nicely.

Arista switches only send MST0 BPDUs on the default VLAN (VLAN1 by default). Because I had first used MST on Cisco switches and had grown accustomed to the way they sent BPDUs on every VLAN, I was convinced that I had found a bug when my Arista switches didn't behave the same way. When I talked to the folks at Arista about it, they pointed out that there is nothing in the RFCs that specifies BPDUs for MST0 being sent out every VLAN.

So, what does this all mean for you in the real world? Let's take a look. I've built a small lab with three switches (SW1, SW2, and SW3), as shown in Figure 17-3. SW1 has a priority of 8,192, SW2 has a priority of 32,768 (the default), and SW3 (the one in the middle of the drawing) has a priority of 4,096. There is one VLAN configured on all three switches: VLAN 100. The links between the switches are trunks with all VLANs permitted.

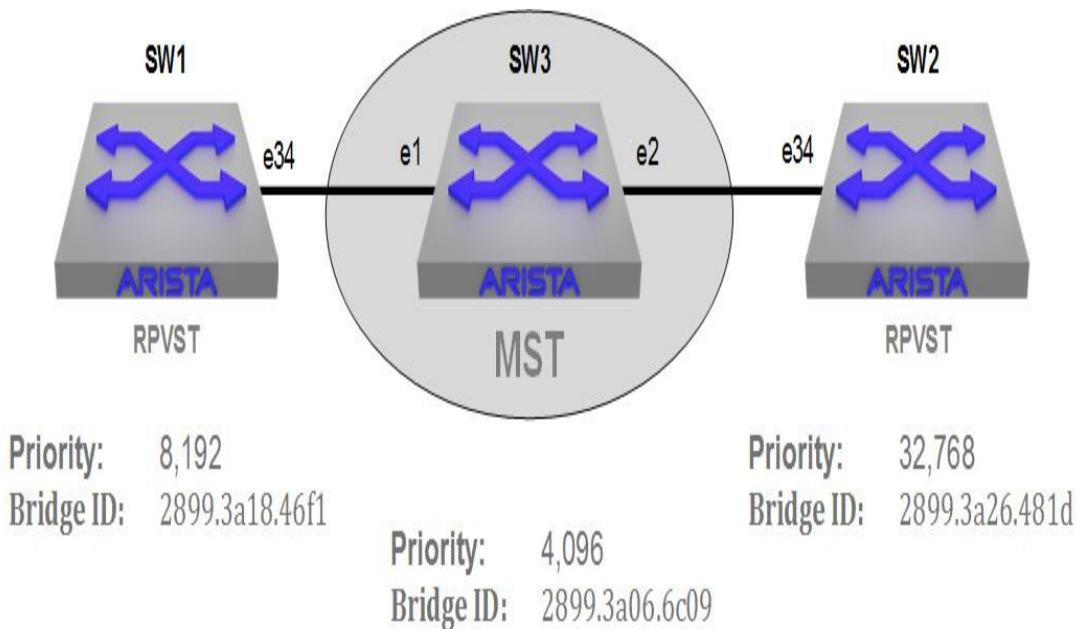


Figure 17-3. RPVST split by MST using Arista switches

If all three of these switches were running Rapid PVST, SW3 would be the root for all VLANs. With MST thrown in the middle, things change a bit, and unless you're careful, the results can surprise you. To further understand this, know that the links between each switch are trunks and that VLAN 100 had been configured on all three switches.

First, let's take a quick look at SW3 in the middle:

```
SW3#sho spanning-tree
MST0
  Spanning tree enabled protocol mstp
  Root ID  Priority  4096
            Address  2899.3a06.6c09
            This bridge is the root

  Bridge ID  Priority      4096 (priority 4096 sys-id-ext 0)
            Address  2899.3a06.6c09
            Hello Time 2.000 sec  Max Age 20 sec  Forward Delay 15 sec

  Interface      Role      State      Cost      Prio.Nbr  Type
  -----  -----
  Et1          designated  forwarding  2000      128.1    P2p Boundary
  Et2          designated  forwarding  2000      128.2    P2p Boundary
```

Looks simple enough. This bridge is the root, and other than it having a priority of 4,096, it appears as if the switch is pretty much in a default configuration. Now let's take a look at SW2, which is running RPVST:

```
SW2#sho spanning-tree
VL1
  Spanning tree enabled protocol rapid-pvst
  Root ID  Priority  4096
            Address  2899.3a06.6c09
            Cost      2000 (Ext) 0 (Int)
            Port      34 (Ethernet34)
            Hello Time 2.000 sec  Max Age 20 sec  Forward Delay 15 sec

  Bridge ID  Priority      32769 (priority 32768 sys-id-ext 1)
            Address  2899.3a26.481d
```

```

Hello Time 2.000 sec Max Age 20 sec Forward Delay 15 sec

Interface      Role      State      Cost      Prio.Nbr Type
-----        -----
-----        -----
Et34          root      forwarding 2000      128.34   P2p

VL100
Spanning tree enabled protocol rapid-pvst
Root ID      Priority      32868
              Address      2899.3a18.46f1
              Cost         2000 (Ext) 0 (Int)
              Port          34 (Ethernet34)
              Hello Time 2.000 sec Max Age 20 sec Forward Delay 15 sec

Bridge ID    Priority      32868 (priority 32768 sys-id-ext 100)
              Address      2899.3a26.481d
              Hello Time 2.000 sec Max Age 20 sec Forward Delay 15 sec

Interface      Role      State      Cost      Prio.Nbr Type
-----        -----
-----        -----
Et34          root      forwarding 2000      128.34   P2p

```

Certainly, the output of the `show spanning-tree` command is different due to this switch running RPVST instead of MST, but look at the VLANs and their respective root bridges. VLAN1 (the default VLAN) shows that SW3 is the root, whereas VLAN 100 shows SW1 to be the root. This is due to the fact that MST does not send out BPDUs on every VLAN, only the default VLAN (VLAN1 in this example).

Now let's look at SW1. Can you guess what it will show?

```

SW1#sho spanning-tree
VL1
Spanning tree enabled protocol rapid-pvst
Root ID      Priority      4096
              Address      2899.3a06.6c09
              Cost         2000 (Ext) 0 (Int)
              Port          34 (Ethernet34)
              Hello Time 2.000 sec Max Age 20 sec Forward Delay 15 sec

```

```

Bridge ID  Priority      32769  (priority 32768 sys-id-ext 1)
          Address       2899.3a18.46f1
          Hello Time   2.000 sec  Max Age 20 sec  Forward Delay 15 sec

Interface      Role      State      Cost      Prio.Nbr  Type
-----+-----+-----+-----+-----+-----+
-----+-----+-----+-----+-----+-----+
Et34          root      forwarding 2000      128.34    P2p

VL100
  Spanning tree enabled protocol rapid-pvst
  Root ID      Priority      32868
  Address       2899.3a18.46f1
  This bridge is the root

Bridge ID  Priority      32868  (priority 32768 sys-id-ext 100)
          Address       2899.3a18.46f1
          Hello Time   2.000 sec  Max Age 20 sec  Forward Delay 15 sec

Interface      Role      State      Cost      Prio.Nbr  Type
-----+-----+-----+-----+-----+-----+
-----+-----+-----+-----+-----+-----+
Et34          designated  forwarding 2000      128.34    P2p

```

SW1 shows that, again, SW3 is the root for VLAN1, but SW1 is the root for all the remaining VLANs! It's almost like RPVST BPDUs were tunneled through MST or something. In a way, that's not far from the truth.

SW3 is the root on VLAN1 for the same reason that SW1 saw SW3 as the root. Because MST operates on the default VLAN, this makes perfect sense.

RPVST (all forms of STP, really) uses a multicast MAC address (01:80:C2:00:00:00) to send BPDUs on all VLANs. Switches not configured for multicast flood these packets out all ports on the respective VLAN, so when SW1 and SW2 send out its BPDUs, SW3

forwards them, but doesn't process them because MST does not listen for BPDUs on VLANs other than the default. The BPDUs from SW1 are then received by SW2, and vice versa, after which each switch on the outer edges processes them due to them running RPVST.

This behavior is different than what you would see with a Cisco switch in the middle. With a Cisco switch running MST in the middle (configured with the same priority of 4,096), the Cisco switch would become the root bridge for all VLANs. This is due to the fact that Cisco enhanced the MST protocol to have this effect.

WARNING

Watch out! If you've designed a network with a Cisco switch in the core running MST and you have PVST switches attached to it, replacing that Cisco core switch with an Arista switch will likely change the STP root bridge on most of the VLANs in your network. If you don't have a root bridge configured for your VLANs in this scenario, they will negotiate one, and it might not be the one that you would otherwise prefer.

Additionally, with an Arista switch in the core running MST, if you add switches to it that are running PVST, either they will become the root for the nondefault VLAN, or they will negotiate with other PVST switches on your network *through* the MST core.

OK, so MST0 works as expected, and it's simple, and we know how to change the root bridge. We've seen how it behaves with RPVST attached, which is interesting, but for many environments, that's good enough. Let's dig in a bit and see how a more complicated scenario might look.

MST0 is only the tip of the iceberg when it comes to MST. Though

MST is not a PVST in the sense that BPDUs are not sent out every VLAN, it has the ability to separate groups of VLANs into separate Spanning Trees. How is that different than PVST? I like to describe it as being a *per-instance* Spanning Tree as opposed to a *per-VLAN* Spanning Tree. Imagine that you have 400 VLANs active on your network. With PVST, you would have 400 Spanning Trees active, each with its own root bridge and each one sending out BPDUs aplenty.

With MST, BPDUs are only ever sent over VLAN1 (by default), but VLANs can be grouped into MST *instances*. For example, you could put VLAN 100 into MST1, 200 into MST2, 300 into MST3, and 400 to 499 into MST4.

With your VLANs grouped like this, you could still do some manual splitting by making one core switch the root for MST1 and MST3, whereas another switch could be the root for MST2 and MST4. As I wrote earlier, I'm not a fan of this type of manual balancing, but I do like to have options, and my customers' desires often override my recommendations, anyway. This is generally a bad idea with MST, though, and the documentation states that there is no benefit to overlapping instances. Because I learn best by breaking things, let's see what happens when we try.

To keep things simple, I use the network in [Figure 17-4](#) for the next examples. This network has only three VLANs—VLAN 100, 200, and 300—all of which are active on all switches and trunked on all links. Note that these are the same switches I used earlier, and I've just changed the connections to form a loop while also configuring all three switches for MST.

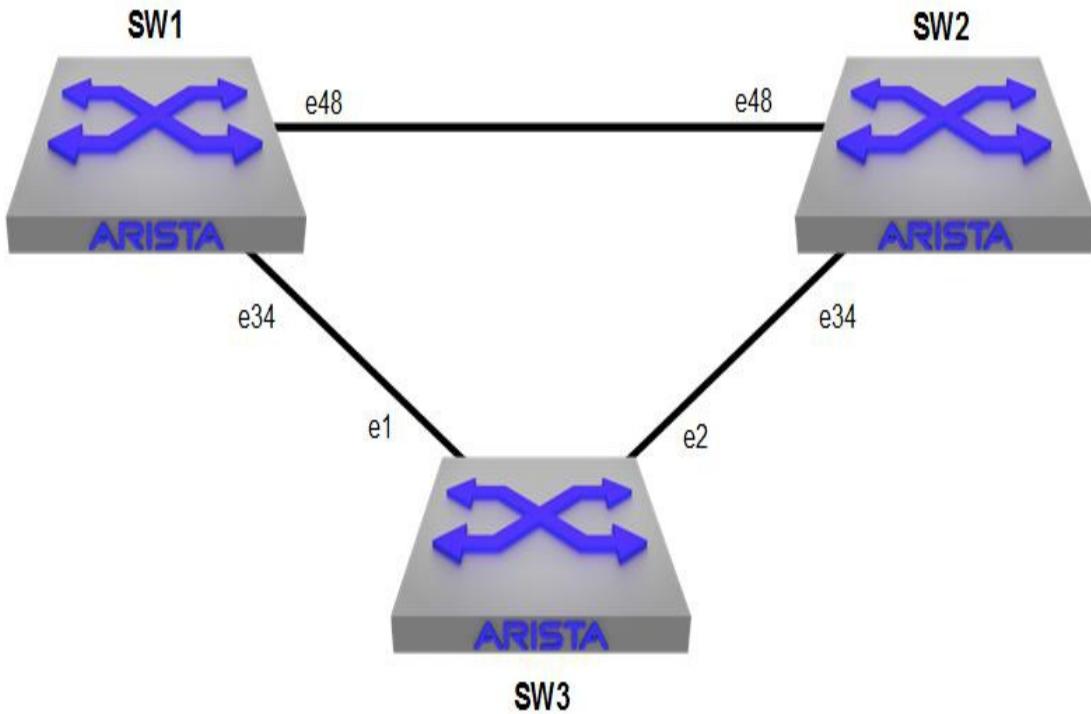


Figure 17-4. Simple MST network

To start things off, I need to put all three switches into MST mode:

```
SW1(config)#spanning-tree mode mstp
SW2(config)#spanning-tree mode mstp
SW3(config)#spanning-tree mode mstp
```

Next, I'm going to create a new MST instance on SW1. Many of the commands for MST are done in the **spanning-tree mst configuration** mode. Like many other such modes, changes do not take effect until you exit the mode:

```
SW1(config)#spanning-tree mst configuration
SW1(config-mst)#
```

To add a new MST instance, use the command **instance**, followed by the *instance number*, and then the VLANs to be included in the instance. To place VLAN 100 into MST instance 1, I use the **instance**

1 **vlan 100** command. Note that VLANs can be listed individually, separated by commas, or listed as ranges:

```
SW1(config-mst)#instance 1 vlans 100
```

Now, I put VLANs 200 and 300 into MST instance 2:

```
SW1(config-mst)#instance 2 vlans 200,300
```

After I exit, **spanning-tree** will be configured, and I should see three instances in MST instead of one:

```
SW1(config-mst)#^z
SW1#sho spanning-tree
MST0
  Spanning tree enabled protocol mstp
  Root ID  Priority  4096
            Address  2899.3a06.6c09
            Cost      2000 (Ext) 0 (Int)
            Port      34 (Ethernet34)
            Hello Time 2.000 sec  Max Age 20 sec  Forward Delay 15 sec

  Bridge ID Priority      8192 (priority 8192 sys-id-ext 0)
            Address      2899.3a18.46f1
            Hello Time 2.000 sec  Max Age 20 sec  Forward Delay 15 sec

  Interface      Role      State      Cost      Prio.Nbr Type
  -----          -----      -----      -----      -----      -----
  Et34          root      forwarding 2000      128.34  P2p Boundary
  Et48          alternate  discarding 2000      128.48  P2p Boundary

MST1
  Spanning tree enabled protocol mstp
  Root ID  Priority  32769
            Address  2899.3a18.46f1
            This bridge is the root

  Bridge ID Priority      32769 (priority 32768 sys-id-ext 1)
            Address      2899.3a18.46f1
            Hello Time 2.000 sec  Max Age 20 sec  Forward Delay 15 sec
```

Interface	Role	State	Cost	Prio.Nbr	Type
<hr/>					
Et34	master	forwarding	2000	128.34	P2p Boundary
Et48	alternate	discarding	2000	128.48	P2p Boundary
MST2					
Spanning tree enabled protocol mstp					
Root ID	Priority	32770			
	Address	2899.3a18.46f1			
This bridge is the root					
Bridge ID	Priority	32770 (priority 32768 sys-id-ext 2)			
	Address	2899.3a18.46f1			
Hello Time	2.000 sec	Max Age 20 sec	Forward Delay 15 sec		
Interface	Role	State	Cost	Prio.Nbr	Type
<hr/>					
Et34 Boundary	master	forwarding	2000	128.34	P2p
Et48 Boundary	alternate	discarding	2000	128.48	P2p

MST0 still sees SW3 as the root from my earlier example, but all of the other instances show SW1 to be the root. This is because the other VLANs are in different MST *regions* on this switch than they are on the other two. An MST region is basically what is formed when one or more switches connect and agree upon their instance mappings. Think of a region as the same instances combining across switches.

This is where MST confuses people, so hang on and let's examine what's really happening. To make things a bit clearer, [Figure 17-5](#) shows a visual representation of what's happening.

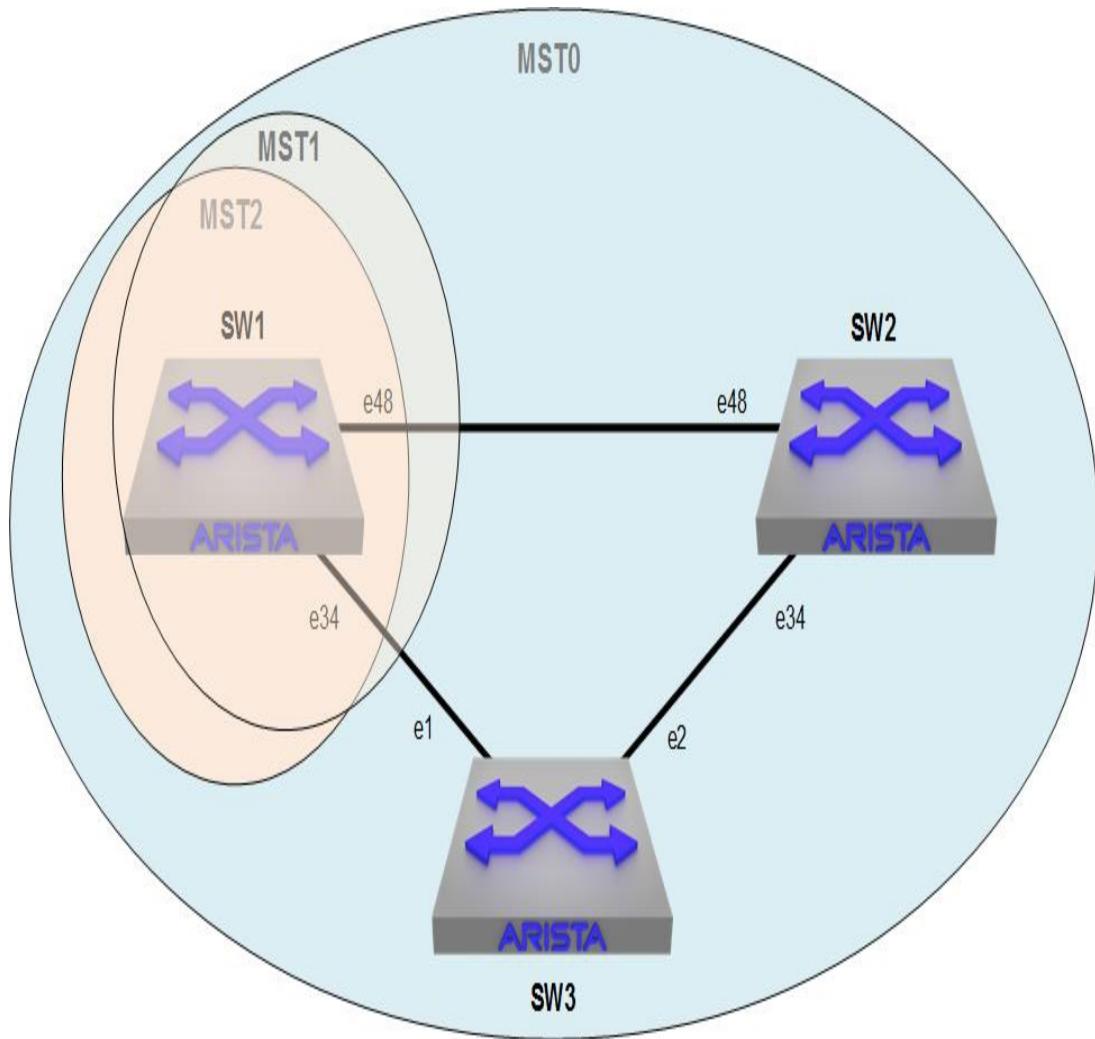


Figure 17-5. MST network with overlapping regions

First, MST0 continues to function the way it always has. MST1 and MST2 are new regions, but they exist only on SW1. Also given the way that MST works, they sort of overlap, as shown in Figure 17-5. If you think that's bad, you're right, but in this example so far it continues to work. What happens to the VLANs that we mapped to MST1 and MST2 on switches SW2 and SW3? Let's take a look. Here's SW2:

```
SW2(config)#sho spanning-tree
MST0
  Spanning tree enabled protocol mstp
```

```

Root ID  Priority  4096
Address  2899.3a06.6c09
Cost     0 (Ext) 2000 (Int)
Port     34 (Ethernet34)
Hello Time 2.000 sec  Max Age 20 sec  Forward Delay 15 sec

Bridge ID Priority  32768 (priority 32768 sys-id-ext 0)
Address  2899.3a26.481d
Hello Time 2.000 sec  Max Age 20 sec  Forward Delay 15 sec

Interface  Role      State      Cost      Prio.Nbr  Type
-----
-----  

Et34        root      forwarding 2000      128.34  P2p
Et48        designated forwarding 2000      128.48  P2p
Boundary

```

And here is SW3:

```

SW3(config)#sho spanning-tree
MST0
  Spanning tree enabled protocol mstp
  Root ID  Priority  4096
            Address  2899.3a06.6c09
            This bridge is the root

  Bridge ID Priority  4096 (priority 4096 sys-id-ext 0)
            Address  2899.3a06.6c09
            Hello Time 2.000 sec  Max Age 20 sec  Forward Delay 15 sec

  Interface  Role      State      Cost      Prio.Nbr  Type
-----
-----  

Et1        designated forwarding 2000      128.1  P2p
Boundary
Et2        designated forwarding 2000      128.2  P2p

```

So, they both see SW3 as the root, just like SW1 did, but they show only MST0. There are some other interesting things going on here, so let's look at what's really, *really* happening.

You might see that some of the interfaces show up as *boundary ports*.

A boundary port is one that connects one MST region to another, but looking at SW3 in MST0, why is Ethernet 1 a boundary port when it's connecting to SW1, which also has MST0 configured? This is the part of MST that I think is a bit complicated, so stay with me.

Yes, all three switches have an MST0 configured, and MST0 is where all of the VLANs are mapped by default. However, there is another layer of organization going on that's not necessarily obvious looking at the configurations or the command output. The thing to note is that just because two switches are configured with the same MST instance number, this doesn't mean that they will form into the same region. In fact, if any of the following parameters are different, two switches with the same MST instance will form separate regions (and here's the rub) that will have the same MST instance identifier! The configurations that must match in order for two switches to form a region are the following:

- Instance to VLAN mapping
- Configuration name

Because MST0 on our three switches are configured differently, Figure 17-6 shows what the network actually looks like.

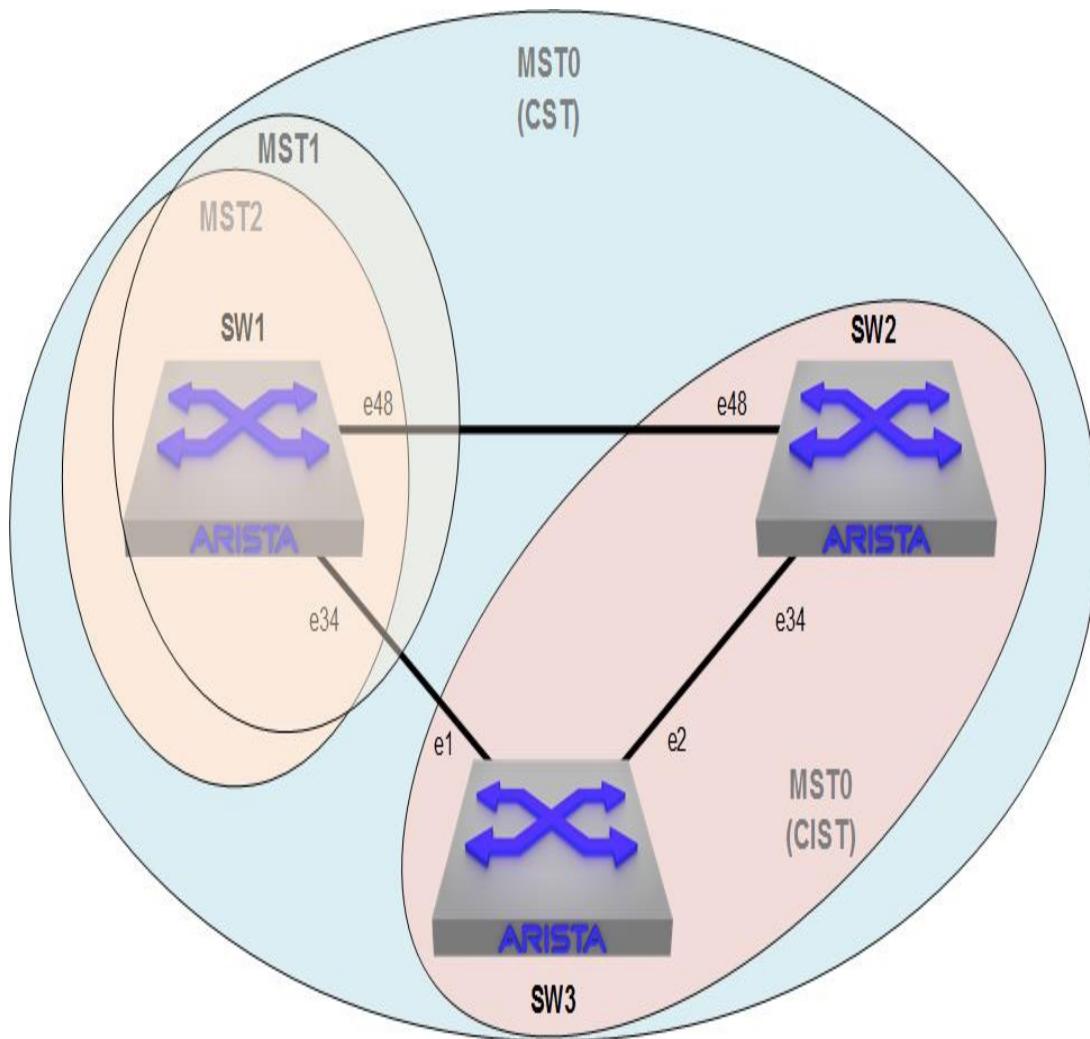


Figure 17-6. Actual MST regions formed based on our configuration thus far

Looking at this frightening diagram we can see that there are actually two *different* MST0 regions! That's why we see boundary ports on our switches in MST0:

```
SW3#sho spanning-tree
MST0
  Spanning tree enabled protocol mstp
  Root ID    Priority    4096
              Address     2899.3a06.6c09
              This bridge is the root

  Bridge ID  Priority    4096 (priority 4096 sys-id-ext 0)
  Address     2899.3a06.6c09
  Hello Time  2.000 sec  Max Age 20 sec  Forward Delay 15 sec
```

Interface	Role	State	Cost	Prio.Nbr	Type
Et1 Boundary		designated forwarding	2000	128.1	P2p
Et2		designated forwarding	2000	128.2	P2p

There are a few common scenarios that will result in boundary ports. First, when a switch configured with Spanning Tree protocol other than MST is attached, the port will be put into a boundary state. Look at the example in [Figure 17-3](#) in which I put an MST switch in the middle of two RPVST switches, and you'll see two boundary ports. These ports are a result of MST being connected to RPVST switches.

This is the kind of thing that can easily be missed by network engineers who aren't seasoned in MST. Again, even though they're all configured with an MST0, they're not all in the same region.

This is explained through another couple of initialisms from the MST lexicon: CST and the CST and the CIST. If you look at [Figure 17-6](#), you'll see that the MST0 that's connecting SW2 and SW3 is labeled as the CIST—this is the CST that's connecting these two MST0s into a region because there was really nothing configured. The larger MST0 is the CST and is what connects all of the regions together, including the MST0 CIST. It can help to think of the CST as sort of the global Spanning Tree. Note that if we had left all three switches with default configurations, they would all be in the same CIST and there would not be the need for a CST. As soon as we connected any of those switches to another region or to a RPVST-enabled switch, boundary ports would form and there would be a CST.

This entire CST/CIST thing is not configured but is part of the way that MST operates. Understanding the way that MST forms regions, though, makes it clearer that there is a sort of hierarchy of Spanning Trees that are built by MST without your intervention. Understanding this hierarchy is the key to understanding MST.

To pound this point home, I'm going to configure SW1 to be the root for all of our new MST instances (but not MST0). This is done in global configuration mode with the `spanning-tree mst instance-number priority priority-value` command. You can also specify the word `root` instead of `priority priority-value`. Here I do both:

```
SW1(config)#spanning-tree mst 1 root primary
SW1(config)#spanning-tree mst 2 priority 8192
```

Another reason that ports will be configured as boundary ports is when MST is running on both switches, but the instances don't match. When attached switches are in the same MST instance, they form a *region*. To be painfully accurate, to form a region, the switches need to have matching instances, with matching VLAN-to-region mapping and the same configuration names. This leads me to one of the main points of confusion (and misconfiguration) I've seen in the field when using MST. Let me show you what I mean, because it's easier to see firsthand. In the next code snippet, I configure SW2 to have the same MST–VLAN mappings, but I add a *configuration name*, which will force the switches to be in different regions, even though they have the same instance numbers:

```
SW2(config)#spanning-tree mst configuration
```

```
SW2(config-mst)#instance 1 vlan 100
SW2(config-mst)#instance 2 vlan 200,300
SW2(config-mst)#name Switch2
SW2(config-mst)#exit
```

Here's how SW2 sees its world with regard to Spanning Tree:

```
SW2(config)#sho spanning-tree
MST0
  Spanning tree enabled protocol mstp
  Root ID  Priority  4096
            Address  2899.3a06.6c09
            Cost      2000 (Ext) 0 (Int)
            Port      34 (Ethernet34)
            Hello Time 2.000 sec  Max Age 20 sec  Forward Delay 15 sec

  Bridge ID Priority  32768 (priority 32768 sys-id-ext 0)
            Address  2899.3a26.481d
            Hello Time 2.000 sec  Max Age 20 sec  Forward Delay 15 sec

  Interface      Role      State      Cost      Prio.Nbr Type
  -----          -----      -----      -----      -----      -----
  Et34          root      forwarding 2000      128.34    P2p
  Boundary
  Et48          alternate  discarding 2000      128.48    P2p
  Boundary

MST1
  Spanning tree enabled protocol mstp
  Root ID  Priority  32769
            Address  2899.3a26.481d
            This bridge is the root

  Bridge ID Priority  32769 (priority 32768 sys-id-ext 1)
            Address  2899.3a26.481d
            Hello Time 2.000 sec  Max Age 20 sec  Forward Delay 15 sec

  Interface      Role      State      Cost      Prio.Nbr Type
  -----          -----      -----      -----      -----      -----
  Et34          master    forwarding 2000      128.34    P2p
  Boundary
  Et48          alternate  discarding 2000      128.48    P2p
  Boundary
```

```

MST2
  Spanning tree enabled protocol mstp
  Root ID  Priority  32770
            Address  2899.3a26.481d
            This bridge is the root

  Bridge ID Priority  32770  (priority 32768 sys-id-ext 2)
            Address  2899.3a26.481d
            Hello Time 2.000 sec  Max Age 20 sec  Forward Delay 15 sec

  Interface      Role     State      Cost      Prio.Nbr Type
  -----
  Et34          master   forwarding 2000      128.34  P2p
  Boundary
  Et48          alternate discarding 2000      128.48  P2p
  Boundary

```

Now SW2 is isolated, and *all* of its interfaces are boundary ports! Let's remove that name and see if we can get it to sync with SW1's config thus forming a region between MST1 and MST2:

```

SW2(config)#spanning-tree mst configuration
SW2(config-mst)#no name Switch2
SW2(config-mst)#^z
SW2#
SW2#sho spanning-tree
MST0
  Spanning tree enabled protocol mstp
  Root ID  Priority  4096
            Address  2899.3a06.6c09
            Cost      2000 (Ext) 2000 (Int)
            Port      48 (Ethernet48)
            Hello Time 2.000 sec  Max Age 20 sec  Forward Delay 15 sec

  Bridge ID Priority  32768  (priority 32768 sys-id-ext 0)
            Address  2899.3a26.481d
            Hello Time 2.000 sec  Max Age 20 sec  Forward Delay 15 sec

  Interface      Role     State      Cost      Prio.Nbr Type
  -----
  Et34          alternate discarding 2000      128.34  P2p Boundary
  Et48          root     forwarding 2000      128.48  P2p

```

MST1

```
Spanning tree enabled protocol mstp
Root ID  Priority  8193
          Address  2899.3a18.46f1
          Cost     2000
          Port     48 (Ethernet48)
          Hello Time 0.000 sec  Max Age  0 sec  Forward Delay  0 sec
```

```
Bridge ID Priority  32769  (priority 32768 sys-id-ext 1)
          Address  2899.3a26.481d
          Hello Time 2.000 sec  Max Age 20 sec  Forward Delay 15 sec
```

Interface	Role	State	Cost	Prio.Nbr	Type	
Et34		alternate	discarding	2000	128.34	P2p
Boundary						
Et48	root		forwarding	2000	128.48	P2p

MST2

```
Spanning tree enabled protocol mstp
Root ID  Priority  8194
          Address  2899.3a18.46f1
          Cost     2000
          Port     48 (Ethernet48)
          Hello Time 0.000 sec  Max Age  0 sec  Forward Delay  0 sec
```

```
Bridge ID Priority  32770  (priority 32768 sys-id-ext 2)
          Address  2899.3a26.481d
          Hello Time 2.000 sec  Max Age 20 sec  Forward Delay 15 sec
```

Interface	Role	State	Cost	Prio.Nbr	Type	
Et34		alternate	discarding	2000	128.34	P2p
Boundary						
Et48	root		forwarding	2000	128.48	P2p

Ah-ha! The status of interface Et48 has gone from **alternate** to **root**, and from **discarding** to **forwarding**, and the type is now **P2p** and no longer **P2p Boundary**. This is what an MST instance should look like when paired with a switch in the same region.

Now that we took care of interface Et48, what's up with Et34?

Remember, Et34 is connected to SW3, and we haven't configured that switch with MST1 or MST2 yet. Because SW2 is configured for MST1, it sees that interface as active in the instance. Because the attached switch on that interface is not configured to be in this instance (and also name and VLAN map), the switch considers it to be in a different **region**, and thus, the port is in a **P2p Boundary** state. Let's go ahead and put SW3 into the same MST configuration, and all of our ports should become state **P2p**.

First, here's the output of `show spanning-tree` on SW3:

```
SW3#sho spanning-tree
MST0
  Spanning tree enabled protocol mstp
  Root ID  Priority    4096
            Address     2899.3a06.6c09
            This bridge is the root

  Bridge ID Priority    4096 (priority 4096 sys-id-ext 0)
            Address     2899.3a06.6c09
            Hello Time  2.000 sec  Max Age 20 sec  Forward Delay 15 sec

  Interface      Role      State      Cost      Prio.Nbr Type
  -----  -----
  Et1           designated  forwarding  2000      128.1    P2p Boundary
  Et2           designated  forwarding  2000      128.2    P2p Boundary
```

So, SW3 is the root for MST0 and has no other configuration yet. It sees both its ports as **P2p Boundary** ports because there are switches attached with different instance VLAN maps. Let's change that by getting SW3 in line with the other two:

```
SW3(config)#spanning-tree mst configuration
SW3(config-mst)#instance 1 vlan 100
```

```
SW3(config-mst)#instance 2 vlan 200,300
SW3(config-mst)#exit
```

Let's see how that's affected the Spanning Tree. Here's the output of the command `show spanning-tree` on SW3 after these changes:

```
SW3(config)#show spanning-tree
MST0
  Spanning tree enabled protocol mstp
  Root ID  Priority  4096
            Address   2899.3a06.6c09
            This bridge is the root

  Bridge ID Priority    4096 (priority 4096 sys-id-ext 0)
            Address   2899.3a06.6c09
            Hello Time 2.000 sec  Max Age 20 sec  Forward Delay 15 sec

  Interface      Role     State      Cost      Prio.Nbr Type
  -----          -----   -----      -----      -----  -----
  Et1            designated forwarding 2000      128.1    P2p
  Et2            designated forwarding 2000      128.2    P2p

MST1
  Spanning tree enabled protocol mstp
  Root ID  Priority  8193
            Address   2899.3a18.46f1
            Cost      2000
            Port      1 (Ethernet1)
            Hello Time 0.000 sec  Max Age  0 sec  Forward Delay  0 sec

  Bridge ID Priority    32769 (priority 32768 sys-id-ext 1)
            Address   2899.3a06.6c09
            Hello Time 2.000 sec  Max Age 20 sec  Forward Delay 15 sec

  Interface      Role     State      Cost      Prio.Nbr Type
  -----          -----   -----      -----      -----  -----
  Et1            root      forwarding 2000      128.1    P2p
  Et2            designated forwarding 2000      128.2    P2p

MST2
  Spanning tree enabled protocol mstp
  Root ID  Priority  8194
            Address   2899.3a18.46f1
```

```

Cost          2000
Port          1 (Ethernet1)
Hello Time   0.000 sec  Max Age  0 sec  Forward Delay  0 sec

Bridge ID   Priority   32770  (priority 32768 sys-id-ext 2)
Address      2899.3a06.6c09
Hello Time   2.000 sec  Max Age 20 sec  Forward Delay 15 sec

Interface   Role      State      Cost      Prio.Nbr Type
-----
-----  

Et1          root      forwarding 2000      128.1    P2p
Et2          designated forwarding 2000      128.2    P2p

```

Huzzah! SW3 is still the root for MST0, but it sees SW1 as the root for MST1 and MST2. All ports are in a P2p state, and there are no boundary ports in any instances. Let's take a look at MST1 on SW1 and then on SW2:

```

SW1#sho spanning-tree mst 1
##### MST1    vlans mapped:    100
Bridge      address 2899.3a18.46f1  priority      8193 (8192 sysid 1)
Root        this switch for MST1

Interface   Role      State      Cost      Prio.Nbr Type
-----
-----  

-----  

Et34         designated forwarding 2000      128.34    P2p
Et48         designated forwarding 2000      128.48    P2p

```

No boundary ports anymore!

Here's the output from SW2:

```

SW2#sho spanning-tree mst 1
##### MST1    vlans mapped:    100
Bridge      address 2899.3a26.481d  priority      32769 (32768 sysid 1)
Root        address 2899.3a18.46f1  priority      8193 (8192 sysid 1)

Interface   Role      State      Cost      Prio.Nbr Type

```

Et34	alternate	discarding	2000	128.34	P2p
Et48	root	forwarding	2000	128.48	P2p

And again, this switch has no more boundary ports. This is the way MST should look when all your switches are configured to be in the same region.

WARNING

Remember that just because you see the correct MST instance on your switch, don't automatically assume that it's in the same region. If you see P2p Boundary ports in your instances, you might have a misconfigured switch. For switches to be in the same region, they must have matching instance VLAN maps, the same configuration name, and the same configuration revision number.

MST Terminology

Let me take a moment and describe how all of these instances and regions interact. To help us along, let's define some terms. I've already used most of these terms in the chapter, so most of this should be apparent by now:

Instance

This is a group of VLANs mapped into a single Spanning Tree. Instance numbers can be within the range of 1 to 4,096. There can be as many as 16 instances configured on a switch.

MSTI

This is a technical term for any MST instance other than instance 0.

Region

A region is one or more switches (technically, bridges) connected

together that are configured with the same VLAN instance maps, the same configuration name, and the same configuration revision (all of these are configured within the `spanning-tree mst configuration` mode).

CST

The Common Spanning Tree is what we've seen as MST instance 0 (MST0). The CST is the means whereby regions are interconnected.

IST

Internal Spanning Tree is used on instance 0 (MST0, or the CST) for the inter-instance Spanning Tree. Consider it to be the Spanning Tree that connects all of the regions together.

CIST

The Common and Internal Spanning Tree comprises all regions, along with the CST (MST0).

It took me a long time to get this all straight, likely because I've spent too much of my life thinking about girls and guitars and not enough time studying, but I've come up with some comparisons that work for me and seem to help other people, as well.

To put all of this stuff into perspective, take a look at [Figure 17-7](#). In this drawing, the large oval represents the *CIST* because it contains everything within it. Within the CIST, there are three smaller circles. These circles each represent an MST *region*. The final configuration from my example earlier in this chapter resulted in a single region. Before there was a single region, there were multiple regions due to the mismatched instance VLAN maps and the temporarily entered configuration name. Where regions connect with other regions, they do so at CIST boundary ports, which we saw in the earlier example.

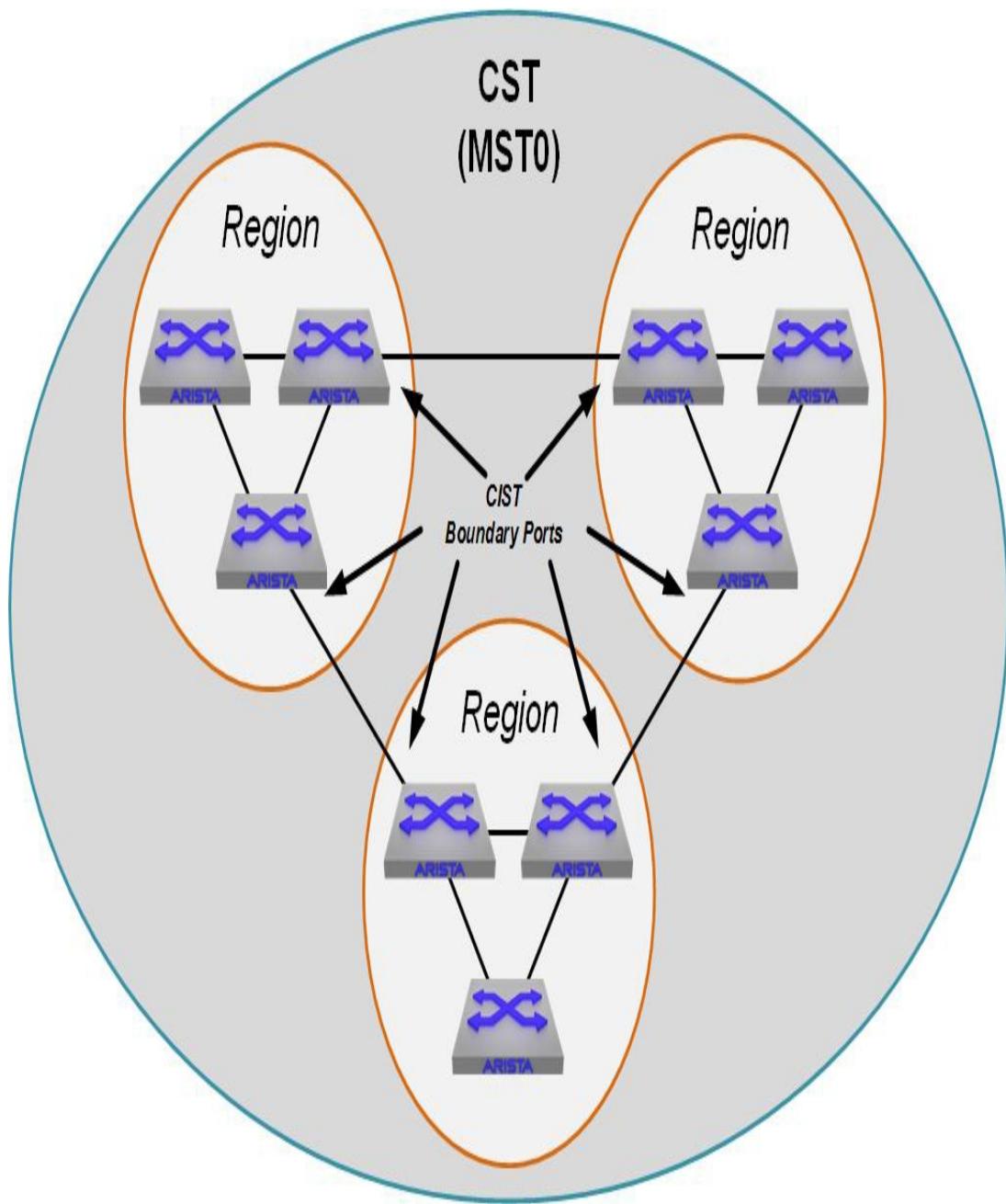


Figure 17-7. Multiple MST regions interconnected with MST0

If you're familiar with Border Gateway Protocol (BGP), each region is sort of like a BGP confederation, in that the region behaves like a single bridge in relation to other regions. Because each region has its own root, the topology within the region is distinct from the topology outside of the region. The topology between regions is distinct from the

topology within regions. I like to call each of the regions a *super switch* when teaching people who are not familiar with BGP, which is completely inaccurate, but it gets the point across. The way the CST (MST0) sees the regions shown in Figure 17-7 is represented in Figure 17-8.

One of the huge benefits of MST is that each region becomes its own failure domain. I used this behavior to solve an environment that had three data centers all bridged together with dark fiber. These three data centers had been designed into a loop and all of the inter-data center links were configured as *trunk all*. It was a nightmare and STP was reconverging constantly. By implementing MST and making each data center its own region, the inter-data center topology changes stopped altogether because each data center had its own Spanning Tree. In MST parlance, each data center was its own region and thus had its own CIST. This provided a huge stability boost in the environment.

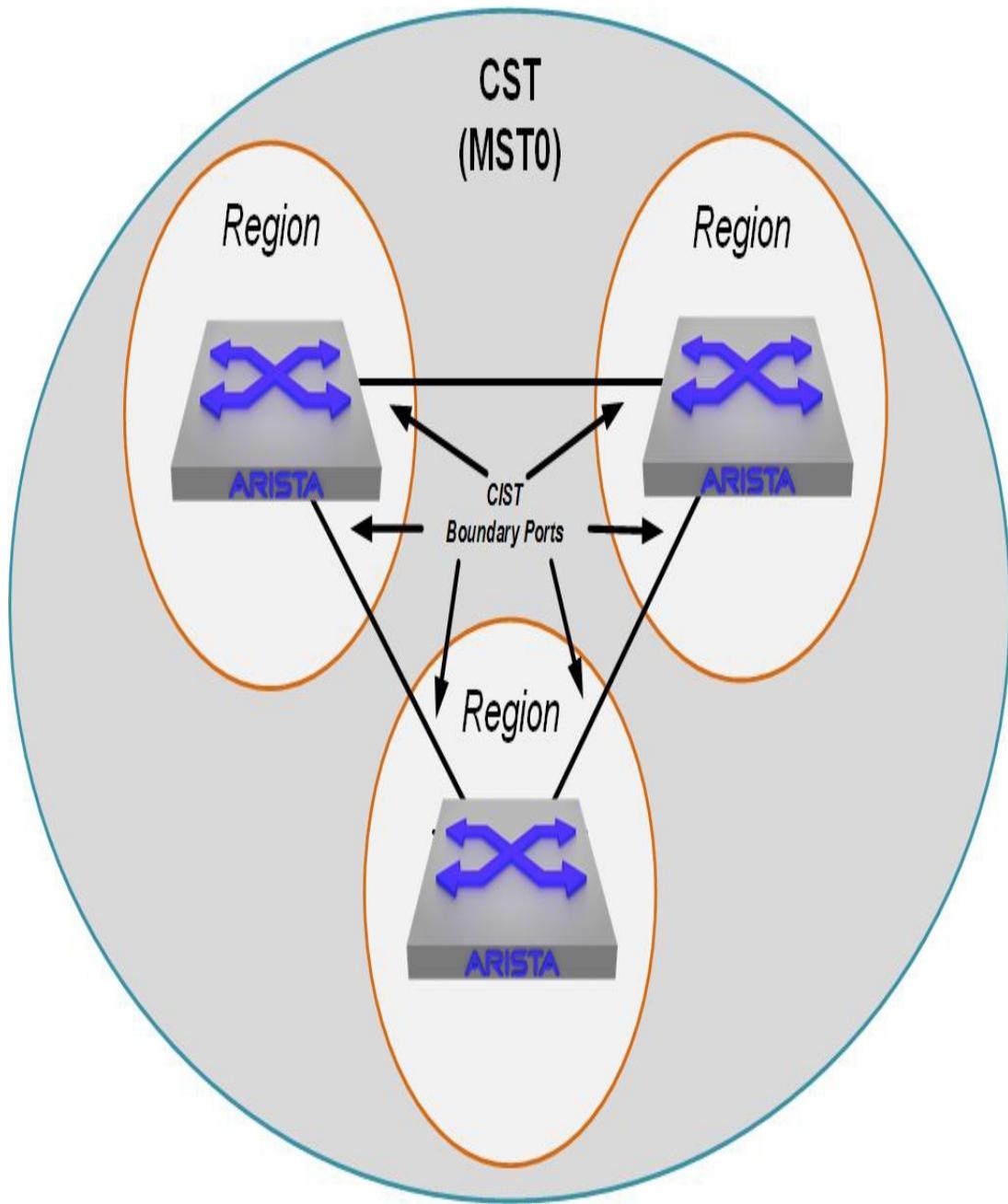


Figure 17-8. Regions as seen by the CST (MST0)

As I warned earlier, it's easy to configure switches into their own region without realizing it. When I design an MST-based network, I like to make each instance into its own region where possible, because this makes it easier for junior engineers to visualize what's going on. Because the regions aren't really identified in the output of the

`spanning-tree` commands, I like to separate using very logical groupings such as data center, row, floor, or something easy to understand. Now, this might not be possible in your network, especially if you're a fan of balancing even/odd VLANs between core switches. Still, so long as the instance VLAN mapping remains consistent and the configuration names and revisions match, getting your regions mapped should be relatively straightforward. That having been said, here's some advice that will help when using MST:

- As Einstein said, make it as simple as possible, but no simpler. The more complicated you make your MST design, the more difficult it will be to understand.
- There aren't really benefits to running multiple regions unless you have a very complicated network. Where I have physically disparate networks in the same building, I'll put them into different instance numbers, so that if they ever do become connected, they will form boundary ports and won't just extend the region.
- One of the ways I like to use regions is when L2 networks span physical buildings. I have a client that has VLANs spanning four buildings. By limiting the interbuilding connectivity, and making each building its own region, the building itself becomes a *super switch* and any improper new links between buildings will affect only the CST topology and not the topology within the building.
- If your network is small, consider a single MST instance with a single region. Though you shouldn't keep everything in MST0, placing all of your switches and all of your VLANs into MST1 isn't necessarily a bad thing..

Pruning VLANs with MST

This little tip is one of those things that people seem to learn the hard way. Let's see if I can help you avoid the pain and misery that can occur when you inadvertently split VLANs with MST. Don't ask me how I know about the pain and suffering. You'll just have to take my word on that part.

[Figure 17-9](#) shows a simple network that looks a lot like the one I've used previously in this chapter. In this network, there are three switches and two VLANs. PC-A, connected to SW2, is on VLAN 300, as is PC-B, which is on SW3. The links between the switches are trunks, but the link between SW2 and SW3 does not have VLAN 300 allowed.

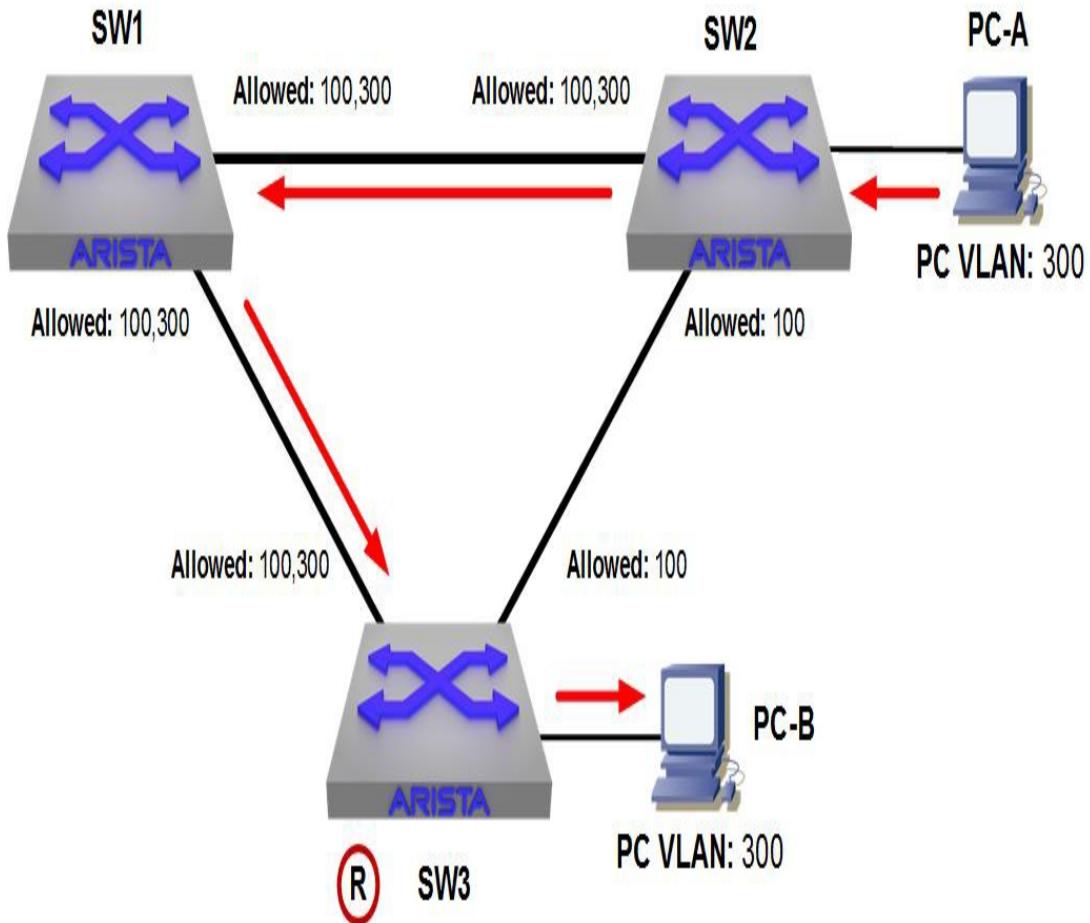


Figure 17-9. Pruned trunks in a PVSTP network

The point of this lesson is to remember that MST does not care about VLANs, only instances, and instances generally contain multiple VLANs. In this network, with PVST, VLAN 300 has no loop, but VLAN 100 does, so only VLAN 100 has a blocked port. That's OK, because there is an alternate path, and all traffic can flow on all VLANs.

Figure 17-10 shows the same network with MST running on all switches. In this example, all switches are configured properly in MST1, and they are all in the same region. Because MST doesn't care about VLANs and only loops within regions, the link between SW1

and SW2 will be blocked, which will cause PC-A to no longer have a path to PC-B. We have a special term for this in networking: it's called *bad*.

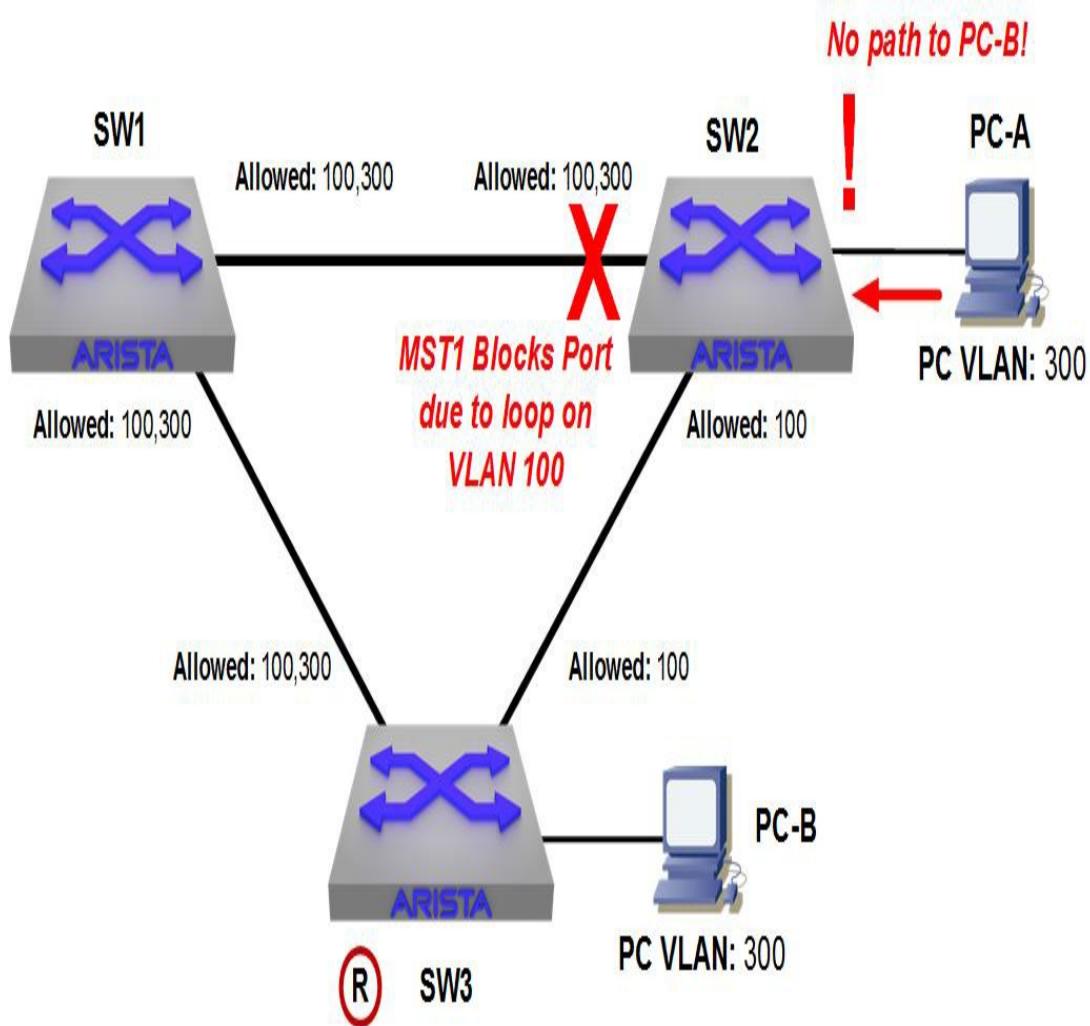


Figure 17-10. Simple, broken network thanks to my trunk-pruning ineptitude

There are a couple of ways to prevent this from happening. If all of these switches truly belong in the same region, you could stop pruning VLAN 300 from the SW2–SW3 link. Pruning can be a great way to limit traffic on links, but in this network, I doubt that VLAN 300 really needed to be pruned in the first place. That's of little comfort, though, when it worked before you converted to MST and now it doesn't. It

turns out that some executives are really sensitive about things like networks working after a change.

Another way to resolve this problem is to put VLAN 100 in one MST instance and VLAN 300 in another. Then, the VLAN 100 instance would block the SW1–SW2 link as shown, but the VLAN 300 instance would have no ports blocked, since that instance would not see a loop. Watch out for overly complex instance VLAN maps, though, because making things complicated leads to switches in different regions when you’re not careful about matching MST configurations.

Conclusion

The Multiple Spanning Tree Protocol is a next-generation Spanning Tree that you should absolutely get to know. Even though it can seem more complex, in practice it isn’t, provided that you design the network with logic and simplicity in mind. And really, shouldn’t all networks be designed that way?

Chapter 18. MLAG

Multichassis Link Aggregation (MLAG) is the open-standard (and thus, Arista) term for linking a port-channel or Link Aggregation Group (LAG) to multiple switches instead of just one. The technology accomplishes the same basic goal as Cisco's Virtual Port Channel (vPC), although, in my experience, MLAG is simpler to configure and more forgiving when used.

MLAG Overview

As just mentioned, the acronym LAG is an abbreviation for *Link Aggregation Group*, which is an open-standard way of describing the bonding of multiple physical links into a single logical link. In Cisco parlance, this technology is called *Etherchannel*. Different vendors use different terms for similar solutions, but the term LAG has become a cross-vendor acceptable way of describing the idea. Why would you want to do this? Let's take a look.

With a traditional network design, interconnecting three switches at Layer 2 (L2) results in a loop. Loops are bad, so Spanning Tree Protocol (STP) blocks the interface on the link farthest from the root. Figure 18-1 shows an example of this.

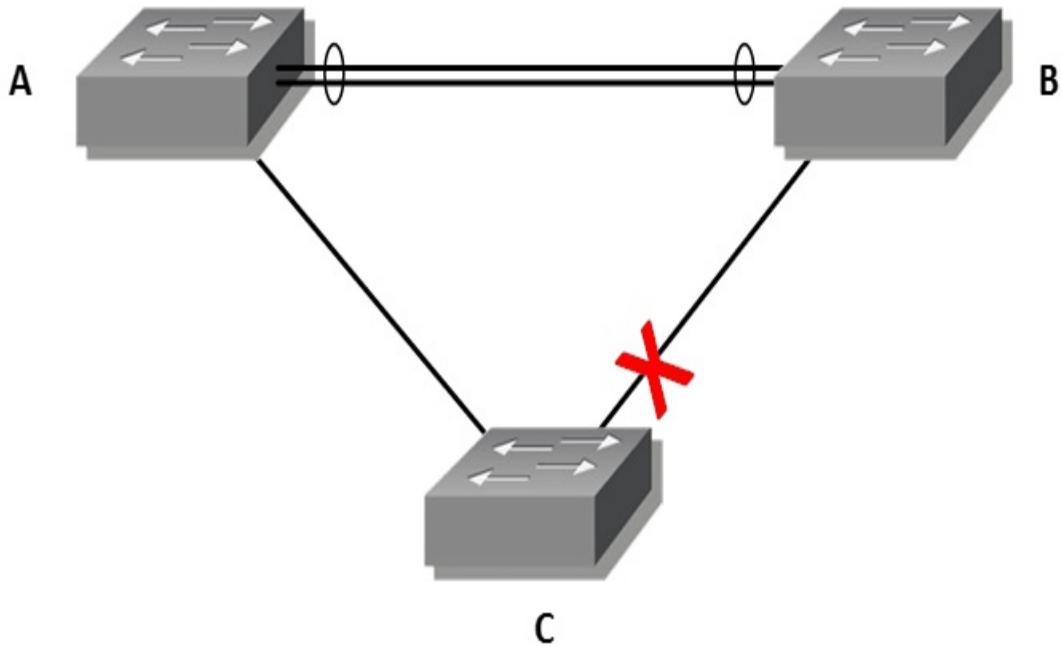


Figure 18-1. Traditional STP-blocked network loop

In this scenario, there is a LAG connecting switch A to switch B. Switch C connects to both A and B switches, forming a loop. STP has blocked the interface on switch C that leads to switch B in order to break said loop. This design will allow for failover if the link between switches A and C were to fail, but the failover can take 30 seconds or more (substantially less if rapid STP is used). Not only that, but only one-half of the available bandwidth to and from switch C is available for use. Wouldn't it be nice if we could use that extra link? Even better, if we used LAG technology, a single link failure wouldn't incur an outage because the second link would already be active.

With MLAG, two Arista switches fool the third switch (or any other Link Aggregation Control Protocol [LACP]-capable device) into thinking that it is connected to a single device. In other words, two Arista switches appear to be one Arista switch to LACP, as shown in Figure 18-2.

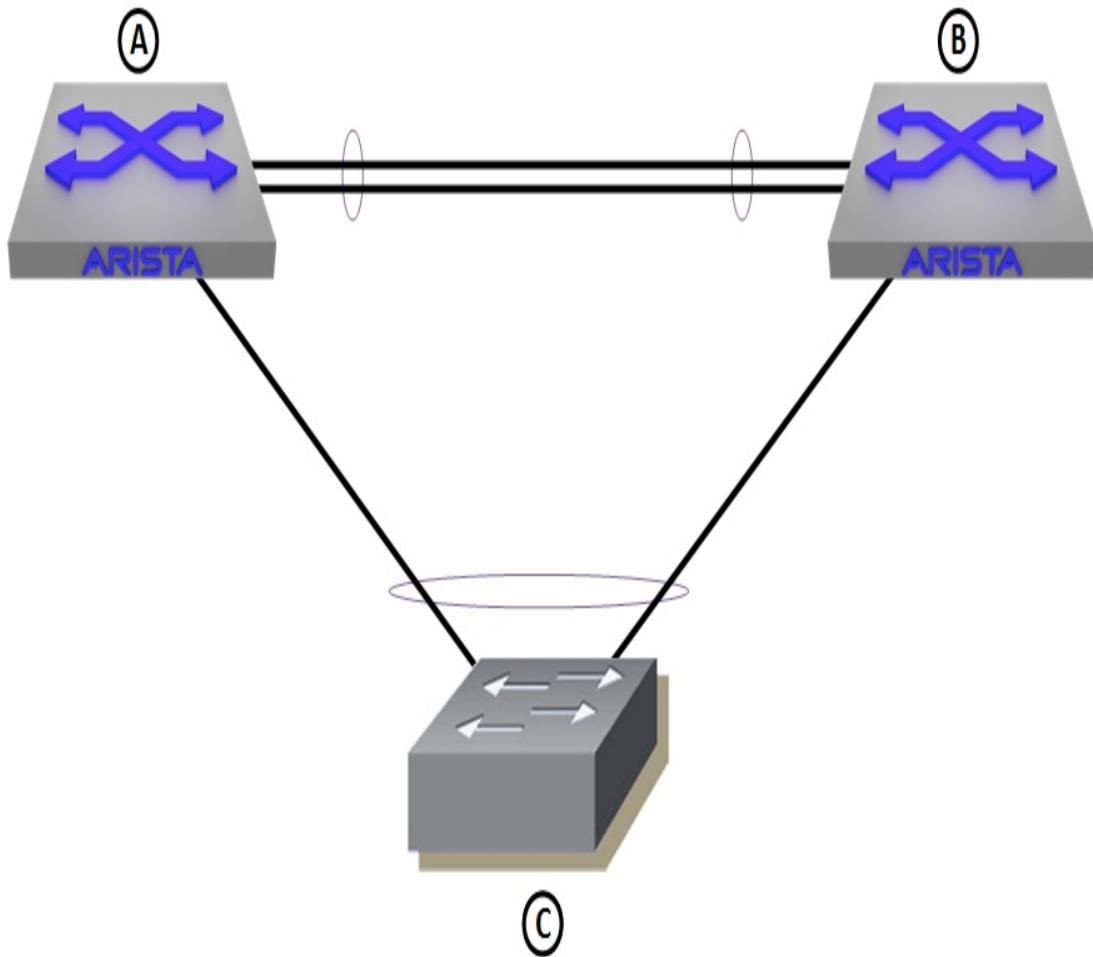


Figure 18-2. Simple MLAG design

With MLAG active using two 10 Gbps links, switch C sees a 20 Gbps logical interface to a single device, even though it is connected to two devices. Arista accomplishes this feat by advertising the same *chassis-ID* from both switch A and switch B. To do this, switch A and switch B must communicate over the A–B switch link, which must be configured with a VLAN that acts as a *peer-link*.

MLAG is configured within something called an *MLAG Domain*. The *MLAG Domain ID* identifies the switch to another switch that will share MLAGs. Let's go ahead and build an MLAG pair.

Configuring MLAG

The first thing we need to do is make sure that both MLAG peers are on the same revision of code. Will it work if the switches have different code? Yes, but in a more limited fashion than it used to. Today if you try to peer incompatible revs of EOS, the switches will report an error and refuse to peer. I'm not a fan of this, but from a TAC point of view, it greatly lowers the number of possible permutations that they need to support. To determine what versions are compatible with what versions, look in the release notes.

This isn't as bad as it might seem at first glance. Looking at the table illustrated in [Figure 18-3](#), EOS 4.21.1F will pair with EOS 4.14.16M, so that's not so bad. The deal is generally that the last or most current revision of code will be supported from each major release, so if you're upgrading from 4.14.5M to 4.21.1F, you won't need to go from 4.14 to 4.15 to 4.16, and so on. You will need to go from 4.15.5M to 4.15.16M, and from there you can go to 4.21.1F. Note also that this is strictly an MLAG compatibility issue and has nothing to do with upgrading standalone switches. You can absolutely upgrade right from 4.14.5M straight to 4.21.1F on a single switch that is not part of an MLAG pair. I'd test that in a lab environment first to see how it might affect your environment, but there is no limitation in the software outside of MLAG.

Release 4.21.X MLAG ISSU Table

EOS Version	Mlag ISSU compatible EOS versions
4.21.0F	4.14.16M, 4.15.10M, 4.16.14M, 4.17.9M, 4.18.8M, 4.19.0F-4.19.3F, 4.19.4M-4.19.9M, 4.19.6.3M, 4.20.1F-4.20.6F, 4.20.7M
4.21.1F	4.14.16M, 4.15.10M, 4.16.14M, 4.17.10M, 4.18.9M, 4.19.0F-4.19.3F, 4.19.4M-4.19.10M, 4.19.6.3M, 4.20.1F-4.20.6F, 4.20.7M-4.20.9M, 4.21.0F
4.21.2F	4.14.16M, 4.15.10M, 4.16.14M, 4.17.10M, 4.18.10M, 4.19.0F-4.19.3F, 4.19.4M-4.19.10M, 4.19.6.3M, 4.20.1F-4.20.6F, 4.20.7M-4.20.10M, 4.21.0F-4.21.1F
4.21.3F	4.14.16M, 4.15.10M, 4.16.14M, 4.17.10M, 4.18.10M, 4.19.0F-4.19.3F, 4.19.4M-4.19.11M, 4.19.6.3M, 4.20.1F-4.20.6F, 4.20.7M-4.20.11M, 4.21.0F-4.21.2F

Figure 18-3. EOS 4.21.1F MLAG ISSU compatibility matrix

You can check the MLAG ISSU compatibility between an installed image and another image local to the switch by using the `show mlag issu compatibility image` command. Let's look at an example of one that passes the test and one that does not. First, here's the revision of code running on my switch:

```
Arista#sho ver | grep image
Software image version: 4.19.10M
```

The switch (a 7280R) is running EOS version 4.19.10M. Here are all of the EOS images that I have stored on flash:

```
Arista#dir EOS*
Directory of flash:/EOS*
-rwx 613330599          Oct  8  2018  EOS-4.19.10M.swi
-rwx 638234211          Mar 20 01:57  EOS-4.20.1F.swi
-rwx 700978970          Nov 12  2018  EOS-4.21.1F.swi

3269361664 bytes total (85319680 bytes free)
```

First, I'm going to run the check against version 4.20.1F:

```
Arista#sho mlag issu compatibility flash:EOS-4.20.1F.swi
/mnt/flash/EOS-4.20.1F.swi (4.20.1F) is MLAG ISSU incompatible with
the current image (4.19.10M). A reboot with this image may cause
packet loss. Please consult the release notes to find a compatible
image.
The new image is compatible with these releases, which may also be
compatible with the current version:
EOS-4.16.6M-INT
EOS-4.16.6M
EOS-4.16.7FX-MLAGISSU-TWO-STEP
EOS-4.16.7M
EOS-4.16.8M
EOS-4.16.8FX-MLAGISSU-TWO-STEP
EOS-4.16.9M
EOS-4.16.10M
EOS-4.16.11M
EOS-4.16.12M
EOS-4.16.13M
EOS-4.16.13FX-MLAGISSU-TWO-STEP
EOS-4.17.0F-INT
EOS-4.17.0F
EOS-4.17.1F
EOS-4.17.1.1FX-MDP-INT
EOS-4.17.2F
EOS-4.17.3F
EOS-4.17.4M
EOS-4.17.5M
```

```
EOS-4.17.6M
EOS-4.17.7M
EOS-4.18.0F
EOS-4.18.0F-INT
EOS-4.18.1.1F
EOS-4.18.2.1F
EOS-4.18.2-REV2-FX
EOS-4.18.3.1F
EOS-4.18.4F
EOS-4.18.4.2F
EOS-4.18.5M
EOS-4.19.0F
EOS-4.19.1F
EOS-4.19.2F
Arista#
```

Well, that certainly threw a lot of output! As I tell network engineers all the time, when you see a page of output, step away from the keyboard and *read it!* In this case, when I first encountered this output, I had to read it probably six times before it sunk in due to an odd bit of technically accurate grammar. Let me run that command again using `grep` to include only the lines that I want to highlight:

```
Arista#sho mlag issu compatibility flash:EOS-4.20.1F.swi |
grep -A3 incompatible
/mnt/flash/EOS-4.20.1F.swi (4.20.1F) is MLAG ISSU incompatible
with
the current image (4.19.10M). A reboot with this image may cause
packet loss. Please consult the release notes to find a compatible
image.
```

The phrase that trips me up is “**is MLAG ISSU incompatible**.” I would probably prefer something like “**is NOT MLAG ISSU compatible**,” or something even simpler like, “**Nope—WON’T WORK!**” but then I suppose that’s why I’m not a developer. When running this command, I look for the list of versions, because if the output of the command spits out a long list of versions, it’s telling you that you should probably use one of those instead of the one you tried.

Let's try that with a different version:

```
Arista#sho mlag issu compatibility flash:EOS-4.21.1F.swi
/mnt/flash/EOS-4.21.1F.swi (4.21.1F) is MLAG ISSU compatible with the
current image (4.19.10M).
```

Not only does it say, “**is MLAG ISSU compatible**,” but there is a notable absence of suggested versions to try instead of the one I checked against. This means that we’re good to go. For the rest of the chapter, both of my switches will be running 4.21.1F, so let’s build a simple MLAG setup using the network shown in [Figure 18-4](#).

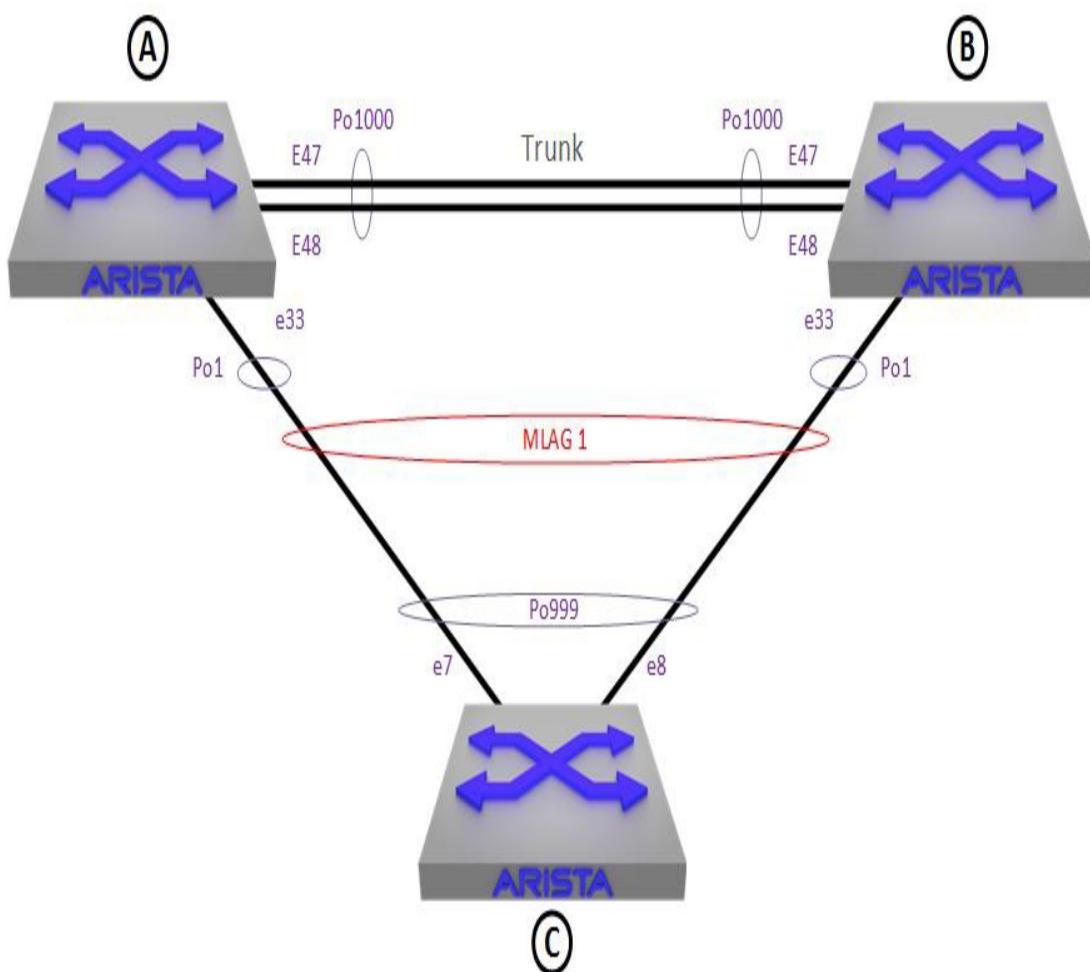


Figure 18-4. MLAG network detail

We need to create a peer-link over which the two switches can communicate. This link can be a single link, but for redundancy, it should always be a port-channel containing a minimum of two physical links. In this example, there are two 24-port switches, so let's use the last two interfaces, e47 and e48:

```
Arista-A(config)#int e47-48
Arista-A(config-if-Et47-48)#channel-group 1000 mode active
```

Next, we configure the port-channel to be a trunk:

```
Arista-A(config-if-Et47-48)#int po 1000
Arista-A(config-if-Po1000)#switchport mode trunk
```

If you're used to Cisco switches, you'll notice that the switch did not bark at us about trunk encapsulation. Here's what would happen on a Cisco switch:

```
Cisco-1(config)#int f1/0/7
Cisco-1(config-if)#switchport mode trunk
Command rejected: An interface whose trunk encapsulation is "Auto"
can not be configured to "trunk" mode.
```

Arista does not negotiate trunk encapsulation, because it supports only dot1q trunks. Older Cisco switches also support Inter-Switch Link (ISL), which is a Cisco proprietary protocol. But enough of my attention deficit issues; let's continue.

Notice that there is absolutely nothing special about this link. It is a port-channel running as a trunk. This is not an MLAG; rather, it's the link used to connect the two peers and, as such, is called the *peer-link*.

With the port-channel configured as a trunk, we need to create a VLAN

that will be used only for MLAG peer-to-peer communication. The Arista examples use VLAN 4094, so let's keep that tradition alive:

```
Arista-A(config)#vlan 4094
Arista-A(config-vlan-4094)#trunk group MLAG-Peer
```

The `trunk group MLAG-Peer` command creates a *trunk group*. A trunk group is a sort of inclusion (or exclusion depending on your point of view) group. When you create a trunk, all VLANs are included on that trunk by default unless you specify otherwise. When we put a VLAN into a trunk group, that VLAN is no longer included in trunks by default. As a result, we now need to assign the same group to the peer-link in order to include that VLAN:

```
Arista-A(config-vlan-4094)#int po 1000
Arista-A(config-if-Po1000)#switchport trunk group MLAG-Peer
```

VLAN 4094 will be included only on trunks that are also assigned to the `MLAG-Peer` trunk group. By doing this, when we create a new trunk, by default VLAN 4094 will *not* be included. This keeps the MLAG peer-link traffic on this link, and only on this link (unless you add the `MLAG-Peer` trunk group to another trunk, but don't do that).

The trunk group names for the peer VLAN should be configured to be the same on both switches. Although they are locally significant, do yourself a favor and keep them the same on the two peers. The configuration for VLANs and VLAN trunk groups must be identical in order to successfully establish an MLAG association between two switches.

Now that we know this VLAN is limited to the peer-link, we can

disable **spanning-tree** on the VLAN:

```
Arista-A(config)#no spanning-tree vlan 4094
```

Note that this is a global command, and not an interface command. It will fail with an **% Incomplete command** message if run from interface configuration mode because the same syntax is used to set cost and port priority there.

Because Multiple Spanning Tree (MST) is the default on Arista switches, and MST is not VLAN based, this command will not have the same result that it would if Rapid-PVST (RPVST) were enabled. It is still a best practice to disable Spanning Tree from the MLAG peer VLAN in case RPVST is ever enabled.

NOTE

Disabling STP is almost always a bad idea. In this case, the MLAG peer-link always needs to be up in order to prevent a *split-brain scenario*. Because the peer-link is using a trunk group, a loop on this VLAN should never occur. The only way a loop could possibly occur would be (in this example) for the MLAG-Peer trunk group to be included on other links from the MLAG pair. So don't do that. Ever.

With the physical link and trunk set up, we're now going to make a Layer 3 (L3) connection between the two switches, as shown in [Figure 18-5](#).

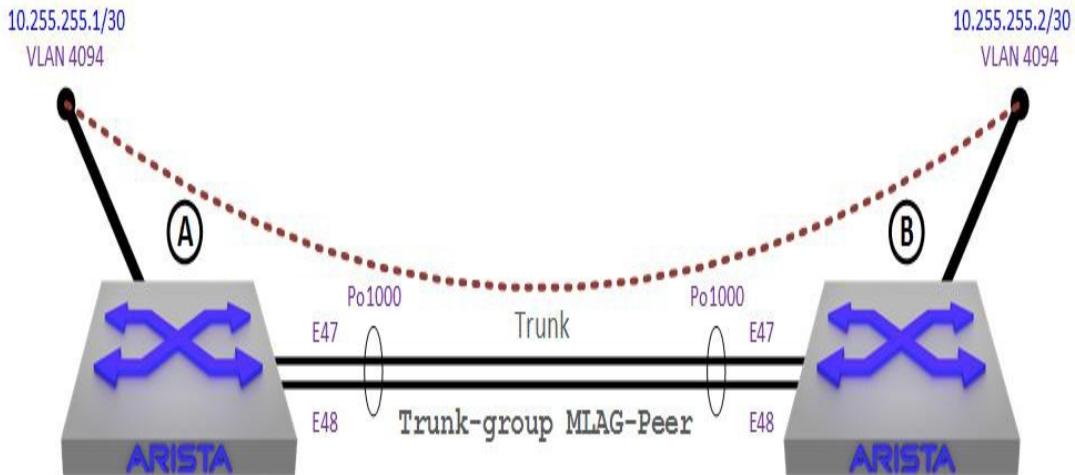


Figure 18-5. MLAG peer-link configuration

Because MLAG peers communicate with each other over L3, we must assign an IP address to the VLAN on each side:

```
Arista-A(config)#int vlan 4094
Arista-A(config-if-Vl4094)#ip address 10.255.255.1/30
Arista-A(config-if-Vl4094)#no autostate
```

The `no autostate` command keeps the L3 Switch Virtual Interface (SVI) interface up regardless of whether there are any interfaces active in the VLAN.

Now, we must configure MLAG itself:

```
Arista-A(config)#mlag
Arista-A(config-mlag)#local-interface vlan 4094
Arista-A(config-mlag)#peer-address 10.255.255.2
Arista-A(config-mlag)#peer-link port-channel 1000
Arista-A(config-mlag)#domain-id Arista-AB
```

The commands should be relatively obvious. We've assigned the MLAG local interface to be the VLAN SVI we just created (VLAN 4094); we've told the switch that the peer for this MLAG domain is at the IP address 10.255.255.2; the peer-link is riding over port-channel

1000; and the MLAG domain ID is Arista-AB (I try to make the domain ID somehow relate to both switch hostnames).

At this point the MLAG peers look like what is shown in [Figure 18-6](#).

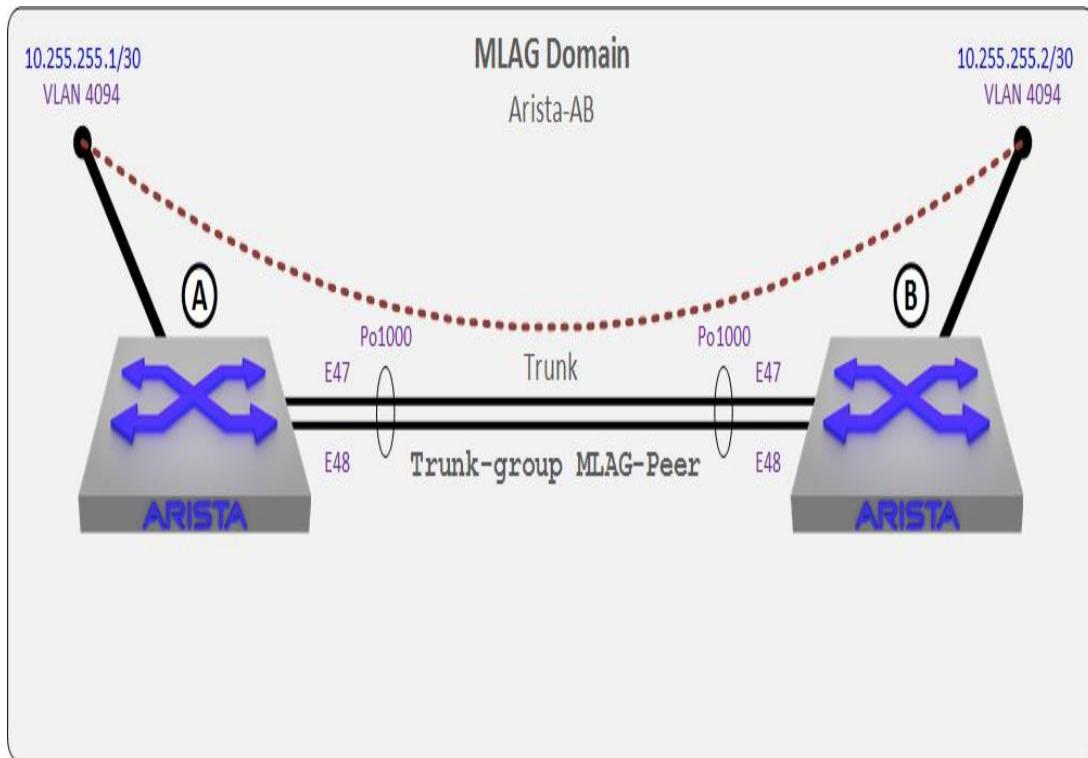


Figure 18-6. MLAG peers with domain ID

The domain ID is the means whereby the switch differentiates different MLAG groups. I show this in more detail later in this chapter. The MLAG domain ID is case-sensitive and must match on both sides.

At this point, the status of the peer-link should be *connected*. This can be shown with the command `show mlag`:

```
Arista-A#sho mlag
MLAG Configuration:
domain-id      :      Arista-AB
local-interface :      Vlan4094
peer-address   :      10.255.255.2
```

```

peer-link          : Port-Channel1000
peer-config        : consistent

MLAG Status:
state              : Active
negotiation status : Connected
peer-link status   : Up
local-int status  : Up
system-id          : 2a:99:3a:06:6f:37

MLAG Ports:
Disabled          : 0
Configured         : 0
Inactive           : 0
Active-partial    : 0
Active-full        : 0

```

The last section that begins with **MLAG Ports** shows all zeroes because we have not created any MLAG interfaces yet, so let's go ahead and create a simple MLAG.

To reiterate, here are the relevant MLAG configurations for both Arista-A and Arista-B:

```

-----
| Arista-A                         | Arista-B
|
-----
| vlan 4094                         | vlan 4094
|
|   trunk group MLAG-Peer          |   trunk group MLAG-Peer
|
|   !
|   !
|   interface Port-Channel1000     |   interface Port-Channel1000
|
|   description [ MLAG Peer-Link ] |   description [ MLAG Peer-Link
]  |
|   switchport mode trunk          |   switchport mode trunk
|
|   switchport trunk group MLAG-Peer |   switchport trunk group MLAG-

```

```
Peer |  
| !  
|  
| interface Ethernet47 | interface Ethernet47  
|  
| description [ MLAG Peer ] | description [ MLAG Peer ]  
|  
| channel-group 1000 mode active | channel-group 1000 mode  
active |  
| ! | !  
|  
| interface Ethernet48 | interface Ethernet48  
|  
| description [ MLAG Peer ] | description [ MLAG Peer ]  
|  
| channel-group 1000 mode active | channel-group 1000 mode  
active |  
| ! | !  
|  
| interface Vlan4094 | interface Vlan4094  
|  
| description [ MLAG Link ] | description [ MLAG Link ]  
|  
| no autostate | no autostate  
|  
| ip address 10.255.255.1/30 | ip address 10.255.255.2/30  
|  
| ! | !  
|  
| mlag configuration | mlag configuration  
|  
| domain-id Arista-AB | domain-id Arista-AB  
|  
| local-interface Vlan4094 | local-interface Vlan4094  
|  
| peer-address 10.255.255.2 | peer-address 10.255.255.1  
|  
| peer-link Port-Channel1000 | peer-link Port-Channel1000  
|  
|  
|-----|
```

By the way, if you think that side-by-side output is cool, that's from an eAPI script I wrote that allows me to compare any command from any

two Arista switches, provided they're running eAPI (and I have the passwords, of course). I use this in my classes all the time for troubleshooting. To learn more about eAPI, see [Chapter 30](#).

In this example, I've set up two Arista switches (Arista-A and Arista-B) connected to a third Arista switch that's been cleverly named Arista-C. The first two Arista switches will be forming an MLAG peer, while the C switch will view the link as a regular port-channel. [Figure 18-7](#) depicts how the network looks before we continue.

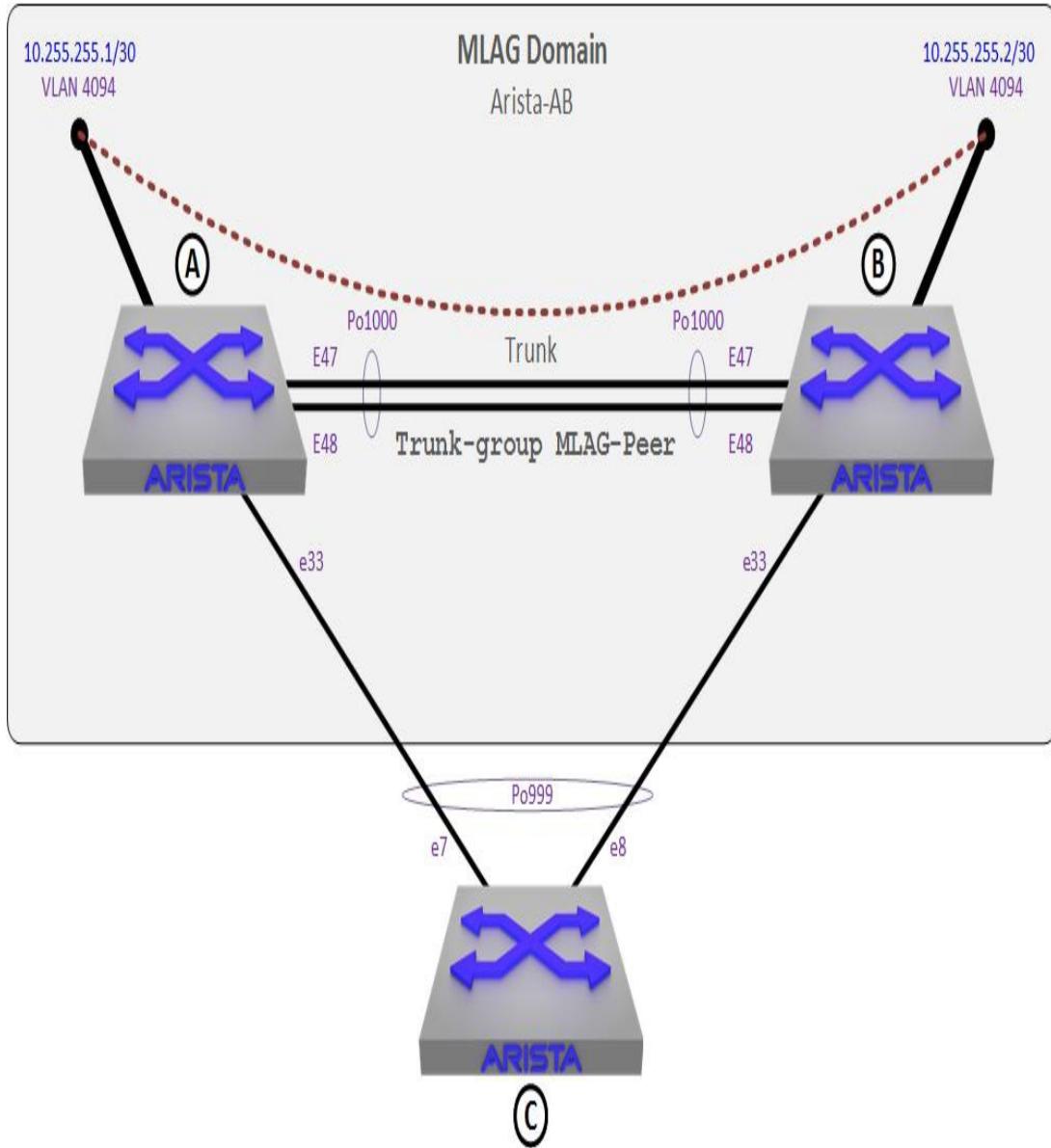


Figure 18-7. Third switch connected via port-channel to MLAG pair

Take careful note that everything we're doing on Arista-C has nothing to do with the MLAG configurations on the two MLAG peers (Arista A and B). This is a very important distinction because nothing MLAG-related “escapes” the MLAG domain. There is no MLAG negotiation outside of the two peers! The only thing Arista-C will see coming from the MLAG peers is LACP.

To further prove that point, here's how I've configured Arista-C:

```
Arista-C(config)#int e7-8
Arista-C(config-if-Et41-42)#channel-group 999 mode active
```

That's it! This switch has absolutely nothing to do with MLAG and has no idea that MLAG is in the mix. The only thing it sees is LACP. To Arista-C, the two MLAG peers appear to be a single chassis.

This forms a simple port-channel (Po999) comprising the physical links, Et7 and Et8. All ports are 10 Gbps. The port-channel will use the LACP protocol due to the `mode active` keywords in the `channel-group` commands.

The problem with the network configuration as it stands is that one of the interfaces in the triangle of network connections will be error-disabled. This is not due to Spanning Tree, but rather LACP on Arista-C, which will receive two different chassis-IDs on E47 and E48. Because those two interfaces are bonded together in a port-channel on Arista-C, LACP expects the remote devices to be a single device. To make that happen, we need to configure the two MLAG peers (Arista A and B) to do that. Luckily, this step is really quite simple.

First, all ports to be bonded between MLAG peers *must* be in a port-channel. You cannot bond physical interfaces, even (as is the case here) if there is only one on each physical switch. Therefore, the first thing we need to do is to put the physical interface on each MLAG peer into a port-channel:

```
Arista-A(config)#int e33
Arista-A(config-if-Et1)#channel-group 1 mode active
```

You must do this on both MLAG peers:

```
Arista-B(config)#int e33
Arista-B(config-if-Et1)#channel-group 1 mode active
```

Do the interfaces and port-channel numbers need to match? No, but do yourself a favor and make them match.

NOTE

I strongly urge you to keep the port-channel assignments the same on the MLAG peers. I've worked on installations where the MLAG peers shared an MLAG using different port-channel interfaces, and it was a nightmare to debug during an outage. Keep it simple, and you'll keep your job.

Now we need a way to bond these two port-channels together across the MLAG pair. To do that, we configure the port-channel itself and apply an MLAG number to the port-channel:

```
Arista-A(config-if-Et1)#int po 1
Arista-A(config-if-Po1)#mlag 1
```

And again, we must do this on both of the MLAG peers:

```
Arista-B(config-if-Et1)#int po 1
Arista-B(config-if-Po1)#mlag 1
```

That's it! After all the peer-link stuff is done and the MLAG adjacency is formed, the creation and linking of port-channels is really all that needs to be done from a daily moves-adds-changes perspective.

Figure 18-8 illustrates what we've built.

It is important to remember that, logically, Figure 18-9 shows how switch C sees the network with MLAG enabled on switches A and B. At this point, switch C has no idea that switches A and B are two different devices, at least so far as LACP is concerned. This is a very important thing to understand because at L3, there are still three devices in the mix. I'm not going to go into a lot of detail on that right now, but remember that MLAG is almost exclusively an L2 thing.

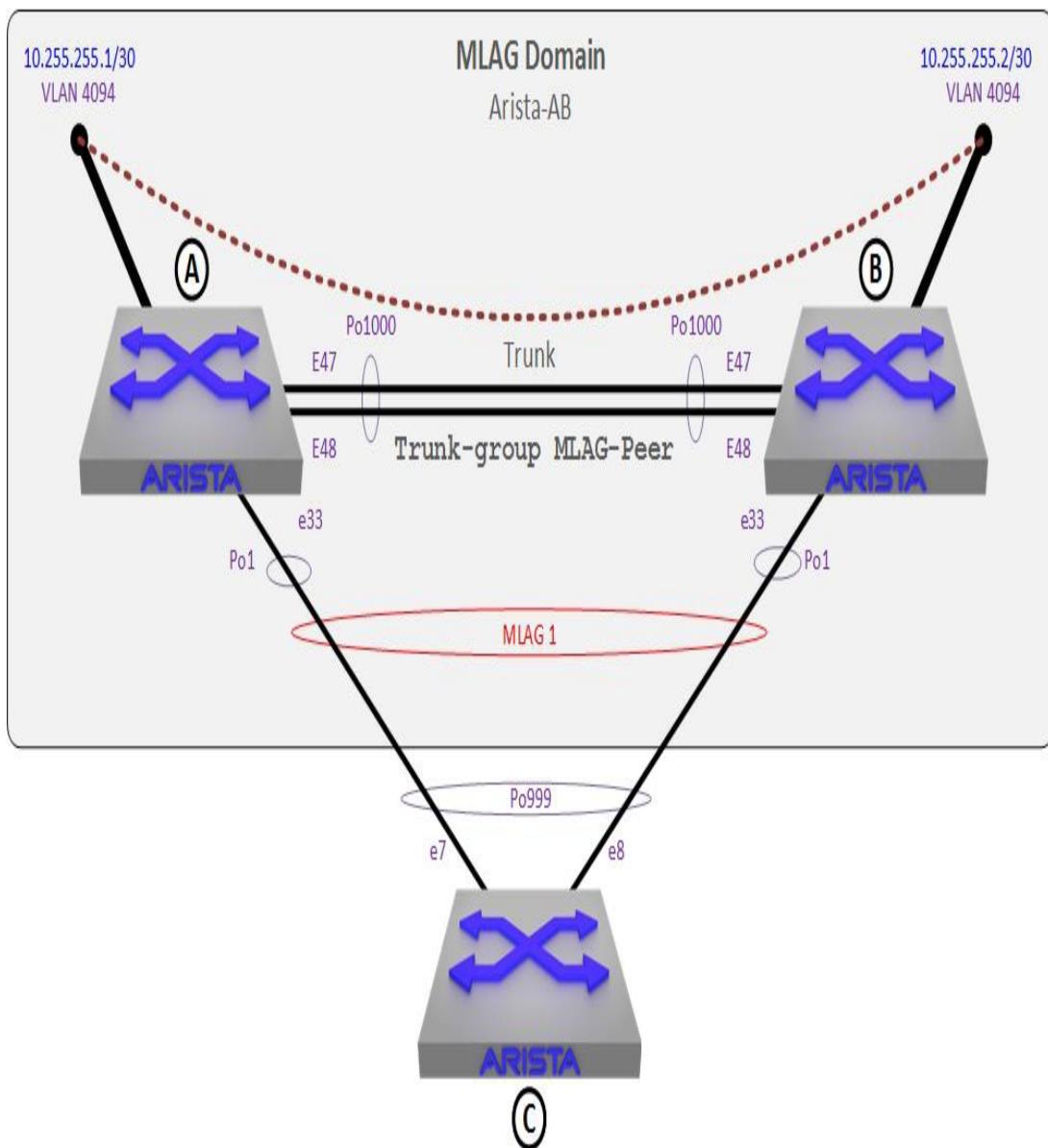


Figure 18-8. MLAG interface added to Arista-A and Arista-B

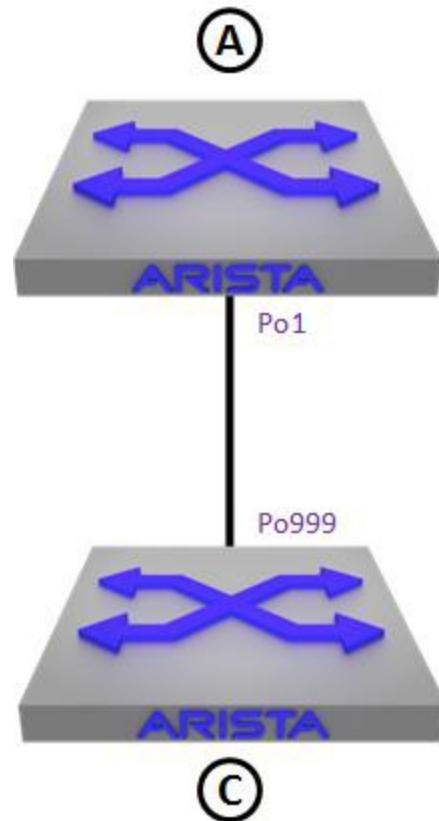


Figure 18-9. How switch C sees the network with MLAG enabled

Managing MLAG

To see the status of individual MLAG interfaces, use the `show mlag interfaces` command:

```
Arista-A(config)#sho mlag int

local/remote
  mlag      desc          state    local      remote
status
-----
---      1      [ Arista-C ]    active-full    Po1      Po1
up/up
```

Here is the output of the same switch with three configured MLAGs, of which only one is active:

```
Arista-A#sho mlag int
```

local/remote	mlag	desc	state	local	remote
status					
-----	-----	-----	-----	-----	-----
up/up	1	[Arista-C]	active-full	Po1	Po1
down/down	3		inactive	Po3	Po3
down/down	5		inactive	Po5	Po5

If MLAG is active, but the peer's link (not the peer-link!) is down for whatever reason, the status of the MLAG will be **Active-partial**:

```
Arista-A#sho mlag int
```

local/remote	mlag	desc	state	local	remote
status					
-----	-----	-----	-----	-----	-----
up/down	1	[Arista-C]	active-partial	Po1	Po1

Here is the same output from the peer with the interface that's down. Check out the local/remote status and how its flipped from the other side because the local interface is always shown first:

```
Arista-B#sho mlag int
```

local/remote	mlag	desc	state	local	remote
status					
-----	-----	-----	-----	-----	-----

1	[Arista-C]	active-partial	Po1	Po1
down/up				

By the way, if you encounter a scenario in which someone has used nonmatching port-channel and MLAG numbers, the `show mlag interfaces` command will be where you'd look to figure it out. Also, smack them in the back of the head for doing that. It's OK. They probably deserve it.

The output of `show mlag` shows you totals as opposed to specific interface information. In this case there is one configured MLAG interface that is active-partial:

```
Arista-A#sho mlag
MLAG Configuration:
domain-id          :      Arista-AB
local-interface    :      Vlan4094
peer-address       :      10.255.255.2
peer-link          :      Port-Channel1000
peer-config        :      inconsistent

MLAG Status:
state              :      Active
negotiation status :      Connected
peer-link status   :      Up
local-int status   :      Up
system-id          :      2a:99:3a:06:6e:0f
dual-primary detection :      Disabled

MLAG Ports:
Disabled           :      0
Configured         :      0
Inactive           :      0
Active-partial     :      1
Active-full        :      0
```

To get some more detail regarding the state of MLAG in general, use the `show mlag detail` command:

```

Arista-A#sho mlag detail
MLAG Configuration:
domain-id      :      Arista-AB
local-interface :      Vlan4094
peer-address   :      10.255.255.2
peer-link      :      Port-Channel1000
peer-config    :      inconsistent

MLAG Status:
state          :      Active
negotiation status :      Connected
peer-link status  :      Up
local-int status  :      Up
system-id       :      2a:99:3a:06:6e:0f
dual-primary detection :      Disabled

MLAG Ports:
Disabled       :      0
Configured     :      0
Inactive       :      0
Active-partial :      1
Active-full    :      0

MLAG Detailed Status:
State          :      primary
Peer State     :      secondary
State changes   :      4
Last state change time :      0:29:21 ago
Hardware ready  :      True
Failover        :      False
Last failover change time :      never
Secondary from failover :      False
Peer MAC address :      28:99:3a:06:6e:ed
Peer MAC routing supported :      True
Reload delay    :      300 seconds
Non-MLAG reload delay :      300 seconds
Peer ports errdisabled :      False
Lacp standby    :      False
Configured heartbeat interval :      4000 ms
Effective heartbeat interval :      4000 ms
Heartbeat timeout :      60000 ms
Last heartbeat timeout :      never
Heartbeat timeouts since reboot :      0
UDP heartbeat alive :      True
Heartbeats sent/received :      22003/22004
Peer monotonic clock offset :      24.884958 seconds
Agent should be running :      True

```

```
P2p mount state changes      :          1
Fast MAC redirection enabled : False
```

MLAG Consistency

You might have noticed that the output of `show mlag` and `show mlag detail` in the previous examples showed that the configuration was *inconsistent*. If you've ever used Cisco's vPC, you know that it can be a bit finicky if things aren't configured properly. In the early days of the Cisco Nexus, I went through some pretty terrible outages due to this behavior, so I was pretty happy to discover that Arista does not enforce *config sanity*. Of course, that can be a pretty sharp double-edged sword, but thankfully Arista has included the means to check config sanity between your MLAG peers using the `show mlag config-sanity` command.

```
Arista-A#show mlag config-sanity
No global configuration inconsistencies found.

Interface configuration inconsistencies:
  Feature      Attribute  Interfaces  Local value  Peer value
  -----  -----
  bridging    access-vlan mlag1        Po1        100      -
```

Somewhere along the line, someone (I bet it was me) configured the MLAG'd port-channel to have the `switchport access vlan 100` command on one side but not the other. Sure enough, here's Arista-A:

```
Arista-A#sho run int po 1
interface Port-Channel1
  description [ Arista-C ]
  switchport access vlan 100
  mlag 1
```

Here's Arista-B:

```
Arista-B#sho run int po 1
interface Port-Channel1
  description [ Arista-C ]
  mlag 1
```

After banging my head on the desk to celebrate my stupidity I removed the offending command and checked again:

```
Arista-A(config)#int po 1
Arista-A(config-if-Po1)#no switchport access vlan
Arista-A(config-if-Po1)#sho mlag config-sanity
No global configuration inconsistencies found.

No per interface configuration inconsistencies found.
```

If you'd prefer to see all the sanity information instead of only what's wrong, you can tack the `all` keyword on the end of the command. Be prepared for some verbosity, though:

```
Arista-A(config-if-Po1)#sho mlag config-sanity all
Global configuration:
  Feature           Attribute      Local value
  Peer value
  -----
  -----
  bridging      admin-state vlan 1      active
active
  bridging      admin-state vlan 5      active
active
  bridging      admin-state vlan 100     active
active
  bridging      admin-state vlan 4094     active
active
  bridging      MLAG-Peer trunk-group vlan 4094   True
True
  lag           lacp port-id range      [0,0]
[0,0]
  lag           lacp system-priority      32768
32768
  mlag          dual-primary-action      dualPriActNone
dualPriActNone
  mlag          dual-primary-detection-delay      0
```

0	mlag	heartbeat-interval	4000
4000	mlag	peer-mac-routing-enabled	False
False	mlag	reload-delay	0
0	mlag	reload-delay lacp mode	False
False	mlag	reload-delay non-mlag	0
0	stp	4094 disabled-vlan	True
True	stp	bpduguard rate-limit interval	0
0	stp	bridge assurance	True
True	stp	forward-time	15
15	stp	hello-time	2000
2000	stp	loopguard	False
False	stp	max-age	20
20	stp	max-hops	20
20	stp	mode	mstp
mstp	stp	mst pvst border	False
False	stp	portchannel guard	False
False	stp	portfast bpdufilter	False
False	stp	portfast bpduguard	False
False	stp	transmit hold-count	6
6			

Interface configuration:

Feature	Attribute	Interfaces	Local value
Peer value			

bridging	access-vlan mlag1	Po1	-
-			
bridging	switchport-mode mlag1	Po1	-

	bridging	trunk-allowed vlan mlag1	Po1	-
	bridging	trunk-native-vlan mlag1	Po1	-
	lag	lacp fallback mlag1	Po1	none
none	lag	lag mode mlag1	Po1	lacp
lacp	vxlan	100 vlan-to-vni	Vx1	10100
10100	vxlan	arp-ip-address-local	Vx1	False
False	vxlan	multicast-group	Vx1	0.0.0.0
0.0.0.0	vxlan	source-interface	Vx1	Loopback1
Loopback1	vxlan	udp-port	Vx1	4789
4789				

I recommend that the `show mlag config-sanity` command be the last step in any change-controls that involve MLAG. It can't hurt, and it might just save your job.

MLAG Failover

I've showed you how to configure MLAG, but I haven't really explained how it all works. Let's fix that.

When you connect two Arista switches by configuring MLAG, the two peers negotiate by comparing their system MAC addresses. The one with the lower system MAC address will be the winner and become the *primary* member of the pair. The loser (there are no trophies for second place in MLAG!) becomes the *secondary* peer. You cannot force this or override this behavior.

The primary peer “owns” the MLAG peer and is responsible for a bunch of stuff (technical term, that) that we’ll get to, but for now understand that the winner of the negotiation uses its system MAC address as the MLAG System ID. The MLAG System ID (MSI) is then used as the chassis-ID that’s sent to the devices that connect to the MLAG domain using port-channels. This is how MLAG tricks those devices into thinking that there is a single device: send an agreed-upon common chassis-ID from two different physical switches.

You can see the system MAC address by using the `show version` command:

```
Arista-A#sho ver | grep MAC
System MAC address: 2899.3a06.6e0f
```

You can see the MLAG System ID by using `show mlag`:

```
Arista-A#sho mlag
MLAG Configuration:
domain-id      :      Arista-AB
local-interface :      Vlan4094
peer-address   :      10.255.255.2
peer-link      :      Port-Channel1000
peer-config    :      consistent

MLAG Status:
state          :      Active
negotiation status :      Connected
peer-link status  :      Up
local-int status :      Up
system-id      :      2a:99:3a:06:6e:0f
dual-primary detection :      Disabled

MLAG Ports:
Disabled      :      0
Configured    :      0
Inactive      :      0
Active-partial :      1
Active-full    :      0
```

If you have an eagle-eye, you might have noticed that the two addresses are actually one bit off. Here I put them next to each other with the `system-id` slightly reformatted so that the numbers line up:

```
2899.3a06.6e0f  
2a99:3a06:6e0f
```

Why this change? This has to do with the fact that the MLAG System ID remains unless both MLAG peers reboot or deconfigure MLAG. This can lead to a rare problem that goes something like this:

Imagine two switches in an MLAG pair. We'll call them A and B. The two switches have the system MAC addresses of aaaa:aaaa:aaaa and bbbb:bbbb:bbbb, respectively. Because the lower MAC address wins, the MLAG System ID (MSI) becomes aaaa:aaaa:aaaa. Now, imagine that the primary switch fails. The secondary switch takes over (more about that in a bit), but the MSI is still aaaa:aaaa:aaaa. Now, let's further imagine that switch A has failed and gets sent back to Arista for service. The new switch that takes its place has a system MAC address of cccc:cccc:cccc. When it joins the MLAG pair, the MSI is *not* renegotiated, and the new switch becomes the secondary in the pair. The MSI is still aaaa:aaaa:aaaa, even though that switch doesn't physically exist in the pair.

Here's where things get weird.

Suppose that the original switch A is returned and is then put in the network, not as one of the original peers, but as a third switch to be connected to the MLAG peers. This new third switch that we'll call

Switch C has a system MAC address of aaaa:aaaa:aaaa. Can you see the problem? Switch C (which used to be switch A, remember) will now try to connect to the MLAG pair using LACP, but LACP will see switch C's own chassis-ID coming in from the MLAG pair and will **err-disable** the port.

To prevent this scenario from happening, the MLAG System ID is actually a derivation of the system-MAC address, and to accomplish that, the MSI is the winner's system MAC address with the locally administered bit set. From Wikipedia:

The second bit of the first byte of a MAC address determines the type of OUI. If the bit is 0 then it is an OUI globally assigned by the IEEE; if the bit is 1 then it is a locally administered MAC address.

To see if your peer is primary or secondary, use the **show mlag detail** command, which also shows you the MLAG System ID:

```
Arista-A#sho mlag detail
MLAG Configuration:
domain-id      :      Arista-AB
local-interface :      Vlan4094
peer-address   :      10.255.255.2
peer-link      :      Port-Channel1000
peer-config    :      consistent

MLAG Status:
state          :      Active
negotiation status :      Connected
peer-link status  :      Up
local-int status  :      Up
system-id      :      2a:99:3a:06:6e:0f
dual-primary detection :      Disabled

MLAG Ports:
Disabled      :      0
Configured    :      0
Inactive      :      0
```

```

Active-partial      :          1
Active-full        :          0

MLAG Detailed Status:
State              :      primary
Peer State         :      secondary
State changes      :          4
Last state change time : 22:05:12 ago
Hardware ready     :      True
Failover           :      False
Last failover change time : never
Secondary from failover :      False
Peer MAC address   : 28:99:3a:06:6e:ed
Peer MAC routing supported :      True
Reload delay       : 300 seconds
Non-MLAG reload delay : 300 seconds
Peer ports errdisabled :      False
Lacp standby       :      False
Configured heartbeat interval : 4000 ms
Effective heartbeat interval : 4000 ms
Heartbeat timeout   : 60000 ms
Last heartbeat timeout : never
Heartbeat timeouts since reboot : 0
UDP heartbeat alive :      True
Heartbeats sent/received : 41441/41441
Peer monotonic clock offset : 24.456001 seconds
Agent should be running :      True
P2p mount state changes :          1
Fast MAC redirection enabled :      False

```

Because the output of `show mlag detail` is so verbose, I'm paring that output down in various ways from this point on because during failover scenarios, it's used a lot, and I don't want this book to be 700 pages. You're welcome.

Again, there is no way to force one side to be primary short of rebooting the primary switch in order to force a failover. For someone like me who likes all of the devices on the left side of my Visio drawings to be active, this is maddening. There is also no command that you can use to force a failover (well, you could reboot one of

them, but that seems excessive). Because I get to work at Arista, I asked the developers why they would deprive me of forced-failover joy, and the answer I received was basically that there is no reason or benefit to having one side be primary over the other. It's taken me years to accept that, but in my experience, it's true. I've moved on and let it go. Mostly.

When the primary switch reboots for whatever reason, the secondary switch becomes primary. Note that the MLAG System ID remains the same. Remember that in my lab that Arista-A was primary, so I went and rebooted it. With it rebooting, I looked at Arista-B:

```
Arista-B#sho mlag det | grep State
State : primary
Peer State : primary
State changes : 9
```

Curious as to why both sides are primary, I asked the developers who said that this is by design because this peer last saw the other peer as primary and assumes that it still is, but because it's lost its connection, it has also assumed the role of primary. When the other side comes up and communicates again, the status will change. Indeed, after the other switch comes back up, we see a better status:

```
Arista-B#sho mlag det | grep State
State : primary
Peer State : secondary
State changes : 9
```

Remember, if Arista-A failed outright and I replaced it with a new switch, there would no longer be a physical switch with that MAC address in the mix, but the MLAG System ID would remain unchanged

unless MLAG is completely deconfigured from both switches in the MLAG domain.

After the Arista-A switch comes back up it remains the secondary switch even though it has the lower system MAC address because *there is no preemption*. Again, it doesn't matter which side is primary, so the switches don't fail over unless there is an outage. Arista does not do preemption because that would just cause more potential network instability, so why force it?

```
Arista-A#sho mlag det | grep State
State : secondary
Peer State : primary
State changes : 2
```

When Arista-A (the original primary that failed) comes back online, all of the interfaces on that switch with the exception of L3 interfaces and the MLAG peer-link pairs are set to `errdisabled`:

```
Arista-A#sho int status
Port      Name          Status      Vlan      Duplex  Speed  Type
Et1
1000BASE-T
Et2
Present
Et3
Present
Et4
Present
Et5
Present
Et6
Present
Et7
Present
[--output removed--]
Et45
Present
Et46
```

Present							
Et47	[MLAG Peer]	connected	in	Po1000	full	10G	
10GBASE-CR							
Et48	[MLAG Peer]	connected	in	Po1000	full	10G	
10GBASE-CR							
Et49/1		errdisabled	1		full	100G	Not
Present							
Et50/1		errdisabled	1		full	100G	Not
Present							
Et51/1		errdisabled	1		full	100G	Not
Present							
Et52/1		errdisabled	1		full	100G	Not
Present							
Et53/1		errdisabled	1		full	100G	Not
Present							
Et54/1		errdisabled	1		full	100G	Not
Present							
Ma1		connected	routed	a-full	a-1G		
10/100/1000							
Po1	[Arista-C]	notconnect	1	full	unconf	N/A	
Po1000	[MLAG Peer-Link]	connected	trunk	full	20G	N/A	

This is to protect your network. If there were something more seriously wrong and this switch were endlessly rebooting, you wouldn't want all of your connected port-channels to bounce and rehash constantly. Think of this as a type of hold-down timer that lets the network stabilize after an outage or planned reboot.

How long do they stay `errdisabled`? The default reload-delay timer is 300 seconds by default for fixed-configuration switches, and 1200 or 1800 seconds for chassis-based switches depending on the hardware platform (starting around EOS 4.21 or so). You can change this behavior with the `mlag configuration` command `reload-delay seconds`. Any value configured below the default will result in a warning when a reload is done (see [“MLAG In-Service Software Upgrade”](#)):

```
Arista-A(config)#mlag configuration
```

```
Arista-A(config-mlag)#reload-delay 120
```

On newer EOS code you can actually define the behavior of non-MLAG interfaces separately from those that belong to an MLAG. A non-MLAG interface is one that does not participate in an MLAG, which I suppose is pretty obvious. The reason for this ability is really to allow L3 interfaces to come up before the L2 MLAG port-channels so that routing protocols can stabilize before MLAGs are rehashed. We do this by using the `reload-delay non-mlag timer` command:

```
Arista-A(config-mlag)#reload-delay non-mlag 60
```

Remember, all MLAG configurations should be the same on both sides:

```
Arista-B(config)#mlag configuration
Arista-B(config-mlag)#reload-delay 120
Arista-B(config-mlag)#reload-delay non-mlag 60
```

You can see how much time is left with the `show mlag` and `show mlag detail` commands during a reload:

```
Arista-A#sho mlag det | grep state
state : Active/Reload (0:01:55 left)
Last state change time : 0:00:22 ago
P2p mount state changes : 1
```

You can see what the configured delay is by using `show mlag detail`:

```
Arista-A#sho mlag detail | grep delay
Reload delay : 120
seconds
Non-MLAG reload delay : 60
seconds
```

You can see whether MLAG is what's holding down your interfaces with the `show mlag detail` command, as well:

```
Arista-A#show mlag det | grep err
Ports errdisabled : True
```

Again, this allows all of the upper-level protocols to stabilize before traffic is forwarded over the links. Additionally, ports don't always come up in the order in which we might expect. For example, the peer-link should always come up first in order for MLAG to work properly, but I always configure the peer-link to be the last ports on the switch. If the switch were to initialize ports in the order in which they are shown in the configuration, the peer-link would come up last. The delay is applied to all non-peer-link ports to prevent that from happening.

Again, you can configure this interval by using the `reload-delay` command within the MLAG configuration, although you should take care when altering this value given that network instability can result when the delay is too short.

NOTE

The time it takes for a switch to finish booting varies based on the number of ports in the switch and the complexity of the configuration. For example, a 7516R with more than a 1,000 ports will take a bit longer to come up than a 7150 with only 24 ports. The 300-second timer value was chosen as a conservative value for a typical 1-rack unit (RU) switch. If you're using chassis switches with hundreds of ports, the value might need to be higher, and Arista recommends 12 minutes (720 seconds) for big chassis deployments.

Remember that the other link in the MLAG interface (e33 on Arista-B

in this example) is up and forwarding traffic during the Arista-A outage. So long as your devices are dual homed to both switches using MLAG, they should stay online while one of the switches in the MLAG pair reboots.

Dual-Primary Detection

Split-brain is the scenario in which the peer-link somehow fails completely and both MLAG peers become primary devices. That's considered bad, though surprisingly in a truly dual-homed environment in which everything is working at L2, it might not be the end of the world. But let's assume that it's bad (because it usually is) and see what we can do to prevent it.

Arista calls a split-brain situation *dual-primary* and has thus created a feature in EOS 4.21 called *dual-primary detection*. This is similar in principle to that other vendor's feature called the *Peer Keepalive Link* in vPC. To configure dual-primary detection you must set the `peer-address heartbeat ip-address mlag` configuration command on each side. Here is the configuration for Arista-A:

```
Arista-A(config)#mlag configuration
Arista-A(config-mlag)#peer-address heartbeat 10.0.0.8
```

Here is the matching configuration on Arista-B. For each switch, these IP addresses are the management interface IPs on the other peer.

```
Arista-B(config)#mlag configuration
Arista-B(config-mlag)#peer-address heartbeat 10.0.0.7
```

With that configured you can also alter the behavior should a dual-

primary state be detected with the dual-primary command. The only real option here is the number of seconds of delay, which you can set from 1 to 1,000 seconds (the last keyword `all-interfaces` in the command has been abbreviated to make it fit on the page). I've configured it the same way on both sides:

```
Arista-A(config-mlag)#dual-primary detection delay 10 action
errdisable all-int
Arista-B(config-mlag)#dual-primary detect. delay 10 action
errdisable all-int
```

To see whether dual-primary is configured, use the `show mlag detail` command:

```
Arista-B#sho mlag detail | grep -i Dual
dual-primary detection      :          Configured
Dual-primary detection delay :          10
Dual-primary action         : errdisable-all
```

With this configured, if the peer-link should go down (you can't shut down the peer-link with interface commands, so it would need to be a hard failure), whichever switch is secondary will take over as primary immediately but will then start dual-primary detection, which basically listens for heartbeats from the configured IP address configured in the `heartbeat` command. It does this only after the delay (if so configured). If dual-primary is detected, it will err-disable all interfaces. When and if the dual-primary state clears, normal MLAG operation should continue.

Bow Tie MLAG

What if you need to connect one MLAG pair to another MLAG pair (or

a pair of Cisco switches using vPC, etc.)? Guess what? Wait for it... nothing changes. Well, you get to use the terrible phrase *Bow Tie MLAG*, so that's something.

Remember, MLAG exists to trick LACP into working. MLAG does not need to be “compatible” with another vendor’s solution because the LACP implementation already works. Cisco’s vPC solution accomplishes much the same thing (though internally in very different ways), so all an Arista MLAG pair should see from vPC is LACP, and all a Cisco vPC pair should see from Arista is, again, LACP.

The two switches on the top (A and B) in [Figure 18-10](#) are an MLAG pair, and the two switches on the bottom (C and D) are an MLAG pair. To connect them together as shown, each pair should have its own MLAG domain ID. Actually, that really doesn’t matter—they can be the same—which is contrary to what I wrote in the first edition. The MLAG domain is locally significant to the MLAG domain (it doesn’t leak out) unless you try to attach a third switch somehow, which isn’t allowed, anyway.

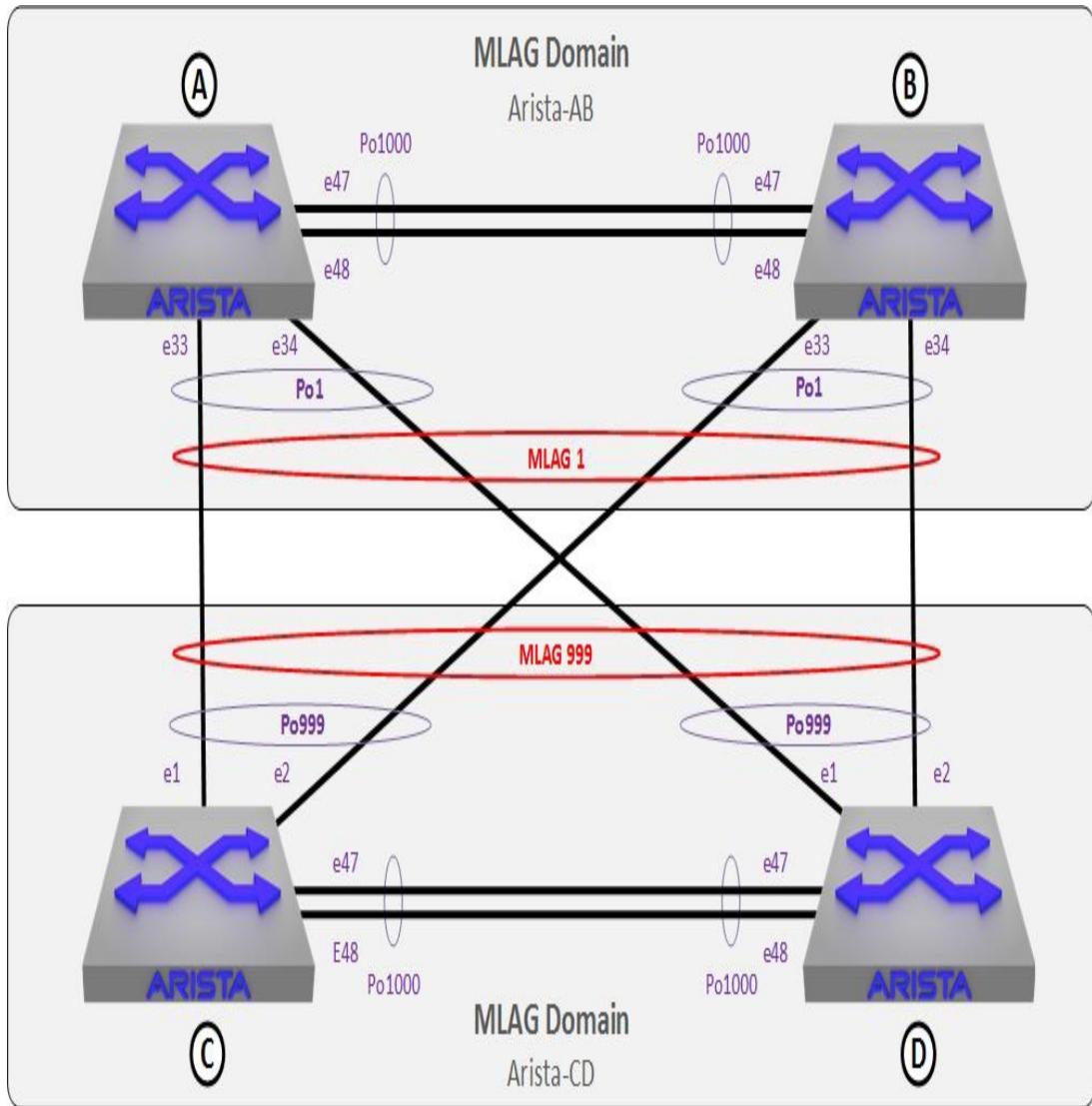


Figure 18-10. Multiple MLAG domain ID

What you'll find if you build this is that it will work if they all have the same MLAG domain. So why require an MLAG domain at all? To make sure that the two configured devices should really be peering. I don't like having multiple pairs with the same MLAG domain name, but I've seen it more than once. Similarly, because the MLAG configuration is local to the peers, I've seen multiple MLAG pairs in an environment using the same IP addresses for the peer-links on each pair! I don't recommend this, but it does seem to work, and I've seen

many customers who have done this. If it were my network, I can tell you that you'd be fixing that, though. While it might work at L2, if you then migrate to an L3 dynamic environment and do something like `redistribute-connected`, you'll get those IP addresses advertised from every pair.

MLAG In-Service Software Upgrade

MLAG In-Service Software Upgrade (ISSU) is a feature enabled on EOS version 4.9.3 and later, and at this point I really hope you're using code that's much later than 4.9.3. With MLAG ISSU, you can upgrade an MLAG switch pair with minimal (subsecond) packet loss and no STP reconvergence. Without MLAG ISSU or if you upgrade while ignoring the switch's dire warnings regarding the state of MLAG ISSU, you'll likely have one or more network topology changes that will result in one or more STP reconvergence events, and no one wants that.

The Arista documentation on MLAG ISSU indicates that the following steps need to be followed in this order to properly upgrade an MLAG ISSU switch pair:

1. Verify primary/secondary state of MLAG on each switch using the `show mlag detail` command, or to be brief, the `show mlag det | grep State` (with a capital “S”) command.
2. Ensure configuration consistencies.
3. Resolve ISSU warnings (from the output of `reload`).
4. Upgrade MLAG secondary switch.

5. Monitor MLAG status using `show mlag detail`.
6. Confirm MLAG secondary status.
7. Upgrade MLAG primary peer switch.
8. Confirm overall MLAG status.

NOTE

When upgrading chassis switch peers that contain dual supervisors, you'll need to upgrade the standby supervisors on both switches, then upgrade the active supervisor on the MLAG secondary, and finally upgrade the last remaining supervisor.

By having switches running MLAG ISSU code, the switches will know whether they can be upgraded without causing an outage. If they cannot, the switch will give you a warning when rebooting. Here's an example of such a warning on a switch running 4.21.1F:

Arista-A#**reload**

If you are performing an upgrade, and the Release Notes for the new version of EOS indicate that MLAG is not backwards-compatible with the currently installed version (4.21.1F), the upgrade will result in packet loss.

Stp is not restartable. Topology changes will occur during the upgrade process.

The following MLAGs are not in Active mode. Traffic to or from these ports will be lost during the upgrade process:

local/remote	mlag	desc	state	local	remote
status					

up/down	1	[Arista-C]	active-partial	Po1	Po1

```
The configured reload delay of 120 seconds is below the recommended
value of 300 seconds. A longer reload delay allows more time to
rollback an unsuccessful upgrade due to incompatibility.
System configuration has been modified. Save? [yes/no/cancel/diff]:
```

As I often joke in my classes, network engineers seem genetically predisposed to being incapable of reading walls of text. If you see a bunch of text like this after typing `reload`, read the damn screen!

WARNING

Using the `reload now` command will cause the switch to bypass these warnings, so don't use the `reload now` command when doing an MLAG ISSU upgrade. This is not meant to be a trick to avoid walls of text, even if that's why a bunch of us do it.

Here's a list of common ISSU warnings and the ways to resolve them.

Compatibility check

The version to which you're upgrading might not be compatible with the version you're on. But then again, it might! Read the release notes to make sure that it is.

STP is not restartable

Usually waiting 30 to 120 seconds will reward you with this warning resolving itself. To see the status of STP restartability (I totally made that word up), use the `show spanning tree bridge detail` command:

```
Arista-A#show spanning-tree bridge detail | inc agent
  Stp agent restartable : True
```

Active-partial MLAG warning

The MLAG shown is not active on the other switch in the MLAG pair. If it should be, bring it up. This is a warning that you'll end up black-holing a device if you continue the reload, so make sure that this is what you're expecting.

Reload delay too low

Remember the reload delay we talked about earlier in this chapter? Well, if the switch thinks that it's too low (lower than the default of 300 seconds for top-of-rack switches and 600 seconds for modular switches), it will bark at you with this warning.

Peer has `errdisabled` interfaces

This is usually an indication that you're impatient and haven't waited long enough for the peer to reboot. Remember, the peer's MLAG-enabled interfaces will stay in an `errdisabled` state for the duration of the reload delay after booting, assuming the other switch is up, and if you're on a switch that shows this warning, that's a good assumption.

The biggest step you should take before considering an MLAG ISSU upgrade is to carefully read the release notes and Transfer of Information (TOI) documents found on the Arista support site. You can find them alongside the EOS binary images. Don't be afraid to call or email your Arista sales engineer or open a TAC case either. Some shops don't do upgrades often enough to remain sharp on the syntax and gotchas, and these folks love to help.

Layer 3 with MLAG

For an example of using Layer 3 with MLAG, check out [Chapter 21](#) which builds an L3 Equal-Cost MultiPathing (ECMP) network including VXLAN terminating on an MLAG pair.

Spanning Tree and MLAG

When MLAG is configured, one of the switches in the MLAG cluster will become the primary switch. The MLAG primary switch will do all of the STP processing, and changes to the secondary will have no effect. There is a pretty big caveat to that statement, though, and that is that changes made to the secondary MLAG switch's STP configuration will be *accepted* to the *running-config*, but they will not take effect unless, that is, the primary MLAG switch relinquishes its role as primary, at which point all of the commands entered on the secondary (now primary) switch will suddenly become active. What's worse, you might not see this coming. Allow me to demonstrate.

I have two switches, Arista-A and Arista-B, configured as an MLAG pair. I have STP left to defaults, and Arista-A is the primary switch in the MLAG domain. I'll be working on Arista-B, so here's proof that it's the MLAG secondary switch:

```
Arista-B(config)#sho mlag detail | grep State
State : secondary
Peer State : primary
State changes : 2
```

And here's the Spanning Tree status:

```
Arista-B(config)#sho spanning-tree
MST0
  Spanning tree enabled protocol mstp
  Root ID  Priority  32768
            Address  2899.3a06.6769
            Cost      0 (Ext) 5999 (Int)
            Port      100 (Port-Channel1)
            Hello Time 2.000 sec  Max Age 20 sec  Forward Delay 15 sec

  Bridge ID Priority  32768 (priority 32768 sys-id-ext 0)
```

Address	2a99.3a06.6e0f
Hello Time	2.000 sec
Max Age 20 sec	
Forward Delay 15 sec	
Interface	Role
State	Cost
Prio.Nbr	Type

Et1	designated
Et34	forwarding
PEt1	designated
PEt34	discarding
Po1	root
	forwarding
	1999
	128.247
	P2p
	Edge
	128.234
	P2p
	128.1
	P2p
	Edge
	128.34
	P2p
	128.100
	P2p

Now, I'll go into that switch (Arista-B) and start mucking with STP. I want to make the priority lower to force it to be the root:

```
SW1(config)#spanning-tree root primary
```

When I make this change, nothing happens:

```
Arista-B(config)#sho spanning-tree | grep Priority
Root ID    Priority    32768
Bridge ID  Priority    32768  (priority 32768 sys-id-ext 0)
```

Frustrated because my change has no effect, I decide to hardcode the priority even lower:

```
Arista-B(config)#spanning-tree priority 4096
```

Huh—still no change:

```
Arista-B(config)#sho spanning-tree | grep Priority
Root ID    Priority    32768
Bridge ID  Priority    32768  (priority 32768 sys-id-ext 0)
```

Beyond frustrated, I start to drink heavily because nothing makes a network change go more smoothly than alcohol.

If I hardcoded the priority to primary (8192) and then 4096, why didn't it show my change? Disgusted and impatient, I rebooted the other switch, because that was so much easier than reading the documentation. Imagine, though, that instead of me rebooting a switch in a lab that these switches were in production, and after my changes didn't work, I gave up and walked away. You know, because that's what happens in real data centers. Anyway, for whatever reason, maybe months later, Arista-A (the primary MLAG switch) reboots. I'll simulate this with a hard reload of Arista-A:

```
Arista-A(config)#reload now

Broadcast message from root@Arista-A (Sat Jan 26 21:20:42 2019):

The system is going down for reboot NOW!
```

All of a sudden and without any real warning, Arista-B is the now root bridge with a priority of 4096:

```
Arista-B(config)#sho spanning-tree | grep Priority
Root ID      Priority      4096
Bridge ID    Priority      4096  (priority 4096 sys-id-ext 0)
```

This happened because Arista-B is now the MLAG primary, as evidenced by the output of `show mlag detail | grep state`:

```
Arista-B(config)#sho mlag det | grep State
State                  :          primary
Peer State              :          primary
State changes           :          3
```

WARNING

The fact that this happens like this is not really a problem; it is functioning by design. The problem is that when configuring STP on the secondary MLAG switch, there are no

warnings that your changes are being saved, and no warnings that any changes made will take effect when and if this switch becomes the primary. Be very careful about making changes to STP when configuring the MLAG secondary switch.

This behavior was recorded on switches running EOS 4.21.1F. When I told Arista about it some six years ago, developers there told me that “the configuration should be the same on both peers.” Um...thanks.

To be fair, the developers have since added the `show mlag config-sanity` command, and had I followed my own advice from earlier in the chapter and issued that command at the end of my change control instead of walking away and not backing out my changes (honestly, I would probably have fired myself if I’d done that), the switch would have told me that I was in danger. Or would it?

Sadly, this is one of the few things that `show mlag config-sanity` does not check. I asked the developers about this, and they said that it was by design without any further explanation. Here’s proof of the fact that it’s not included. First, here’s the relevant configuration from Arista-A:

```
Arista-A#sho run section span
spanning-tree mode mstp
no spanning-tree vlan 4094
```

And here is Arista-B’s relevant configuration:

```
Arista-B#sho run section span
spanning-tree mode mstp
no spanning-tree vlan 4094
spanning-tree mst 0 priority 4096
```

As you can see, Arista-B has a Spanning Tree priority of 4096, which is a big change from the default of 32768 on Arista-A. Here's what `show mlag config-sanity` says on Arista-A:

```
Arista-A#sho mlag config-sanity
No global configuration inconsistencies found.

No per interface configuration inconsistencies found.
```

Here's what `show mlag config-sanity` says on Arista-B:

```
Arista-B#sho mlag config-sanity
No global configuration inconsistencies found.

No per interface configuration inconsistencies found.
```

The lesson to learn here is that the configurations should be the same on both peer switches, and you should always make sure that's the case both with the `show mlag config-sanity` command and something like the `sho run section span` (or similar) command.

Conclusion

One last note, because this comes up a lot: no, you should not disable STP if you're using MLAG (or any vendor's MLAG-like technology). Ask any networking consultant whether they've heard of a Spanning Tree event being caused by someone bringing in a home office switch and connecting it where it didn't belong. I know I've seen that more than once. Hell, I had a client who refused to run more than two Ethernet runs to each cube, insisting that should anyone need more ports, they could just bring in a switch from home. This is an outage waiting to happen, and STP is the last line of defense against the loop-

inducing server guy who needs 14 ports on his desk. Do yourself a favor and outlaw switches on (or under) desks. And keep STP running, because when you outlaw desktop switches, only outlaws will have desktop switches...or something.

MLAG works great if you're in need of a multihomed L2 design. I've taught people who favor end-to-end L3 designs who seem to get angry that MLAG exists, which always kind of amuses me. Arista is not forcing anyone to use any one design over another. If you need an L2 solution, MLAG is great. If you need L3, go for it.

Chapter 19. First-Hop Redundancy

First-hop redundancy is the ability for one or more devices to share the same IP address in order to provide multidevice resiliency in default gateway scenarios (though they can be nondefault gateways, too). Usually, this involves one device owning the IP address while other devices stand by, ready to assume control of the address should the owner fail. This is not always the case, however, as we'll see.

Cisco's proprietary Hot Standby Router Protocol (HSRP) is probably what most Cisco shops are using to accomplish this, but outside of the Cisco world the Virtual Router Redundancy Protocol (VRRP) is the standard. This is likely due to the fact that it is an open source protocol and therefore supported by multiple vendors. Arista's Extensible Operating System (EOS) supports VRRP, but also introduces an interesting new feature called Virtual Address Resolution Protocol (VARP). In this chapter, we take a look at both VRRP and VARP, including configuration examples and reasons why you might choose one solution over the other.

VRRP

If all you've ever used is Cisco's HSRP, don't worry, because VRRP is pretty much the same thing. In fact, it's so similar that Cisco

complained vigorously when the RFC for VRRP was announced. VRRP is defined in RFCs 2338, 3768, and 5798, whereas Cisco's HSRP is defined in RFC 2281. The RFC for HSRP states the following:

2 Conditions of Use

US Patent number 5,473,599 [2], assigned to Cisco Systems, Inc. may be applicable to HSRP. If an implementation requires the use of any claims of patent no. 5,473,599, Cisco will license such claims on reasonable, nondiscriminatory terms for use in practicing the standard. More specifically, such license will be available for a one-time, paid up fee.

Cisco complained to the Internet Engineering Task Force (IETF) with the following:

In Cisco's assessment, the VRRP proposal does not represent any significantly different functionality from that available with HSRP and also implementation of draft-ietf-vrrp-spec-06.txt would likely infringe on Cisco's patent #5,473,599. When Cisco originally learned of the VRRP proposal, the Hot Standby Router Protocol was then promptly offered for standardization with the understanding that, if approved, licenses for HSRP would be made available on reasonable, nondiscriminatory terms for implementation of the protocol.

The full text of Cisco's response to the VRRP RFC is available online. My point in this little legal tangent is that VRRP is *very* similar to HSRP, so it's nothing to be afraid of. But surely, there must be differences; otherwise Cisco would be suing every vendor that uses VRRP. Let's take a look at how to configure VRRP, and we'll see how things differ as we go.

First, we need to get some terms straight. In VRRP, a *virtual router* is one or more devices configured as a sort of cluster, wherein one of the devices is the *master router*, and the rest are *backup routers*. This is a bit different than HSRP, in that HSRP can have only one active router and one standby router. Any other devices configured in the same HSRP group are in a listening state. When an HSRP event occurs that causes both the active and backup routers to fail, a new election takes place among the listening routers.

With VRRP, you can configure multiple routers to be in the virtual router, and they'll all be backup routers if not elected as the master.

NOTE

Even though the primary devices in this book are switches, the term router is used accurately here because VRRP can be configured only on a Layer 3 (L3) interface.

According to RFC 2338, a VRRP virtual router is defined by “A virtual router identifier (VRID) and a set of IP addresses.” In other words, for routers to become part of a virtual router, they must agree on the VRID and IP addresses. The VRID is the group number (just like HSRP), and the IP addresses are the virtual IPs (VIPs) to be shared.

NOTE

A quick note about Arista’s implementation of VRRP is in order. In many vendors’ implementations, the configured IP address of the interface can be used as the VIP. According to the RFC, this is an option, though as of version 4.21.1F this is not yet supported in EOS.

Additionally, if you're building this in a lab, make sure that you have `ip routing` enabled, because although you can configure all of this on your interfaces, the `show vrrp` won't produce any output unless `ip routing` is enabled.

Basic Configuration

Let's begin by looking at a simple network, as shown in Figure 19-1. There are two switches, both connected to VLAN 99. Each switch has a Switch Virtual Interface (SVI) configured and IP routing enabled. The server will be configured to use these switches as its default gateway.

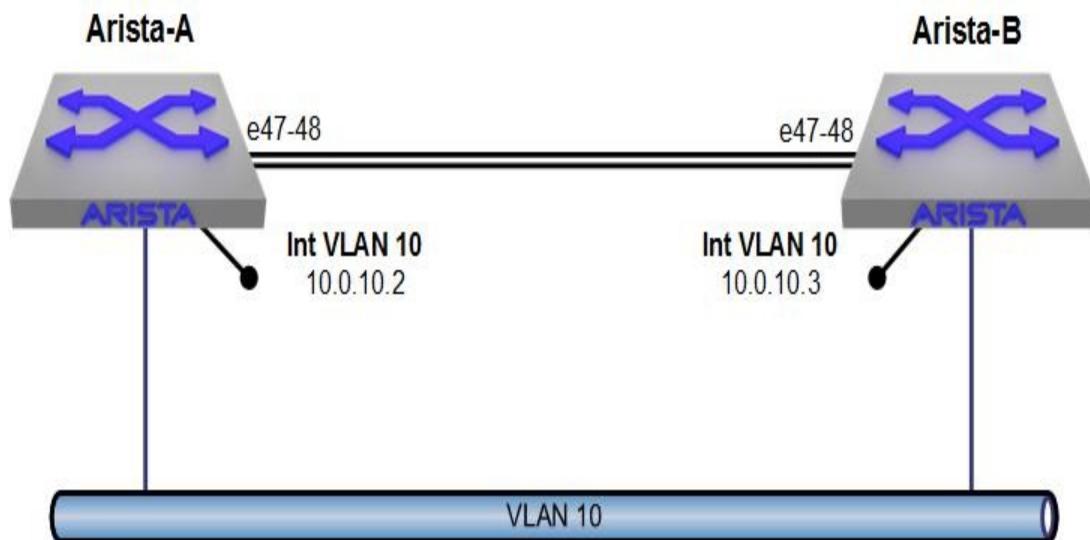


Figure 19-1. A simple network in need of VRRP

We begin with a simple configuration for the VLAN 10 interface on Arista-A:

```
Arista-A(config)#int vlan 10
Arista-A(config-if-Vl10)#ip address 10.0.10.2/24
```

And here's Arista-B:

```
Arista-B(config)#int vlan 10
Arista-B(config-if-Vl10)#ip address 10.0.10.3/24
```

It would be difficult to get much simpler than that! So, let's add the simplest of VRRP configurations. In its simplest form, VRRP just needs a virtual router group number, and a group IP address, which I will add to the A switch:

```
Arista-A(config-if-Vl10)#vrrp 10 ip 10.0.10.1
```

The group number can be any number inclusive of 1 to 255. According to the Arista configuration guide (section 3.1.1), *Two virtual routers cannot be assigned the same VRID, even when they are on different VLANs. A virtual router's scope is restricted to a single LAN.* This can seem misleading, but I think the RFC has the answer:

However, there is no restriction against reusing a VRID with a different address mapping on different LANs. The scope of each virtual router is restricted to a single LAN.

In short, you can have the same group number on multiple interfaces, but you cannot have the same group number/IP address combination on two interfaces, which makes perfect sense because you can't really have the same IP network on two interfaces anyway (VRFs notwithstanding).

At this point, VRRP is active (assuming that there are interfaces active in the VLAN), and we can view the status by using the `show vrrp` command:

```
Arista-A(config)#sho vrrp
Vlan10 - Group 10
  VRF is default
```

```
VRRP Version 2
State is Master
Virtual IPv4 address is 10.0.10.1
Virtual MAC address is 0000.5e00.010a
Mac Address Advertisement interval is 30s
VRRP Advertisement interval is 1s
Preemption is enabled
Preemption delay is 0s
Preemption reload delay is 0s
Priority is 100
Master Router is 10.0.10.2 (local), priority is 100
Master Advertisement interval is 1s
Skew time is 0.609s
Master Down interval is 3.609s
```

Because this is all it takes to configure a virtual router, many people prefer to add the IP address last if they will be altering any of the default values. If you have devices out there looking for this IP address, it just became available, and when we go in and muck with the default values, the status of the new IP might change.

Speaking of defaults, let's take a look at them from the previous output. Without configuring anything, the switch has become the master. That makes sense because there are no other switches participating yet. Note that the *advertisement interval* is one second and that *preemption* is enabled. By comparison, HSRP sends *hello packets* every three seconds, which raises another important difference between the two protocols.

With HSRP, both the active and standby routers send hello packets, whereas any routers configured to listen just, well, listen. With VRRP, only the active router sends out advertisements; the backup routers all just listen.

Preemption is enabled by default with VRRP, which I think is great because, honestly, I can't remember the last time I configured HSRP without preemption enabled. Still, if you'd like to disable preemption, you can do so with the `no vrrp group-number preempt` command:

```
Arista-A(config-if-Vl10)#no vrrp 10 preempt
```

Preemption can also be delayed by using a couple of interesting options, which you can see by using the question mark after the `delay` keyword, as shown here:

```
Arista-A(config-if-Vl10)#vrrp 10 preempt delay ?
  minimum  Specify the minimum time in seconds for the local router to
  wait
          before taking over the active role. Default is 0 seconds.
  reload   Specify the preemption delay after a reload only. This delay
  period
          applies only to the first interface-up event after the router
  has
          reloaded. Default is 0 seconds.
<cr>
```

To set the minimum delay for 30 seconds and the reload delay for 60 seconds, you can combine both into the following single command:

```
Arista-A(config-if-Vl10)#vrrp 10 preempt delay minimum 30 reload
  60
```

The VRRP master router sends out advertisements every one second by default. To change it, use the `vrrp group-number timers advertise seconds` command. Here, I change the default to 10 seconds:

```
Arista-A(config-if-Vl10)#vrrp 10 timers advertise 10
```

Now that I've messed with all the defaults, let's take a look at the output of `show vrrp` again:

```
Arista-A(config-if-Vl10)#sho vrrp
Vlan10 - Group 10
  VRF is default
  VRRP Version 2
  State is Master
  Virtual IPv4 address is 10.0.10.1
  Virtual MAC address is 0000.5e00.010a
  Mac Address Advertisement interval is 30s
VRRP Advertisement interval is 10s
  Preemption is disabled
  Preemption delay is 30s
  Preemption reload delay is 60s
  Priority is 100
  Master Router is 10.0.10.2 (local), priority is 100
Master Advertisement interval is 10s
  Skew time is 0.609s
  Master Down interval is 30.609s
```

OK, that was fun, but let's put all those values back to their defaults so we can move on:

```
Arista-A(config-if-Vl10)#vrrp 10 preempt
Arista-A(config-if-Vl10)#default vrrp 10 timers
Arista-A(config-if-Vl10)#default vrrp 10 preempt delay
```

Let's make sure that everything is back to normal:

```
Arista-A(config-if-Vl10)#sho vrrp
Vlan10 - Group 10
  VRF is default
  VRRP Version 2
  State is Master
  Virtual IPv4 address is 10.0.10.1
  Virtual MAC address is 0000.5e00.010a
  Mac Address Advertisement interval is 30s
VRRP Advertisement interval is 1s
  Preemption is enabled
  Preemption delay is 0s
  Preemption reload delay is 0s
```

```
Priority is 100
Master Router is 10.0.10.2 (local), priority is 100
Master Advertisement interval is 1s
Skew time is 0.609s
Master Down interval is 3.609s
```

Now let's see what happens when we add Arista-B to the mix, with all of the defaults left untouched:

```
Arista-B(config)#int vlan 10
Arista-B(config-if-Vl10)#vrrp 10 ip 10.0.10.1
```

Let's see what Arista-B thinks about our new VRRP configuration:

```
Arista-B(config-if-Vl10)#sho vrrp
Vlan10 - Group 10
  VRF is default
  VRRP Version 2
State is Backup
  Virtual IPv4 address is 10.0.10.1
  Virtual MAC address is 0000.5e00.010a
  Mac Address Advertisement interval is 30s
  VRRP Advertisement interval is 1s
  Preemption is enabled
  Preemption delay is 0s
  Preemption reload delay is 0s
  Priority is 100
  Master Router is 10.0.10.2, priority is 100
  Master Advertisement interval is 1s
  Skew time is 0.609s
  Master Down interval is 3.609s
```

Looks good, and everything has the same default values. Figure 19-2 shows the new VRRP-enabled network.

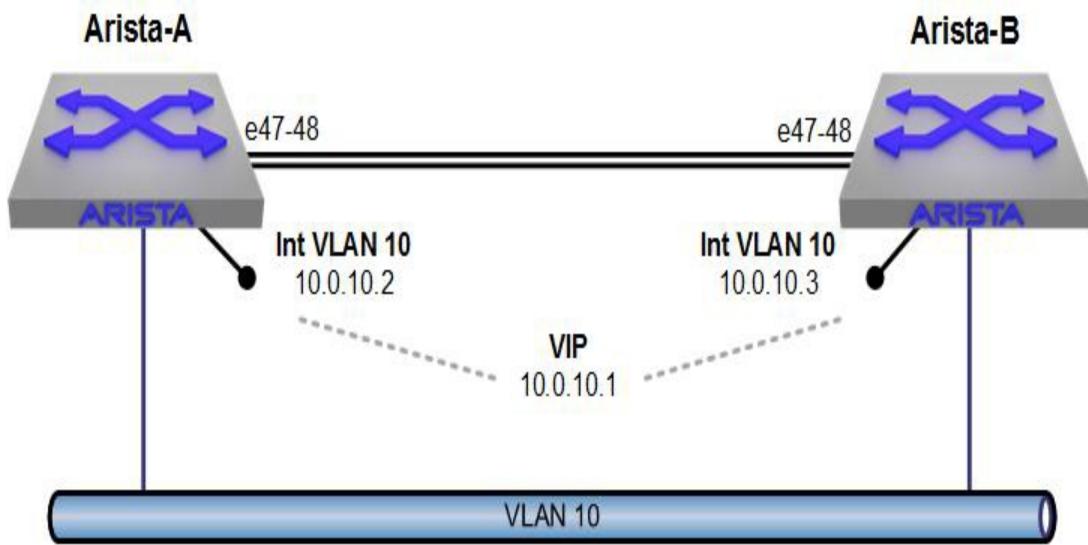


Figure 19-2. VRRP enabled on our simple network

So, what made Arista-B decide to become the backup router? It became the backup because a master already existed and that master had the same priority as Arista-B. Here's what the RFC says on the subject:

The protocol should ensure after Master election that no state transition is triggered by any Backup router of equal or lower preference as long as the Master continues to function properly.

If I were to pull off all of the configurations and configure Arista-B first, it would become the master, and when I then configured Arista-A, it would become the backup. This can be different behavior than that exhibited by HSRP. If two routers are both configured in the same group, on the same VLAN, with the same VIP, they will negotiate who will become the master, and a new master might be chosen (depending on the version of code; this behavior has changed over time).

I like to have a little more control over which switch becomes the master router, so I'll assign a priority to the VRRP group. Because

Arista-B is currently the backup, let's configure it with a higher priority and see what happens. The default priority is 100. The priority value can be any integer in the range of 1 to 254, so let's make Arista-B have a priority of 105:

```
Arista-B(config-if-Vl10)#vrrp 10 priority 105
```

NOTE

Higher priorities are better in VRRP. Well, who's to say what *better* really means? I suppose that I could say that higher priorities are more desirable, but that makes me think that the master routers should be prettier. Instead, I'll write that the router with the highest priority will become the master router, and leave it at that. Also, a priority of 255 indicates that the interface IP is the VIP, which is why doing so will force the router to become the master. Only you can't do that in EOS, so never mind.

The switch quickly becomes the master because it has a higher priority than Arista-A:

```
Arista-B(config-if-Vl10)#sho vrrp
Vlan10 - Group 10
  VRF is default
  VRRP Version 2
State is Master
  Virtual IPv4 address is 10.0.10.1
  Virtual MAC address is 0000.5e00.010a
  Mac Address Advertisement interval is 30s
  VRRP Advertisement interval is 1s
  Preemption is enabled
  Preemption delay is 0s
  Preemption reload delay is 0s
  Priority is 105
  Master Router is 10.0.10.3 (local), priority is 105
  Master Advertisement interval is 1s
  Skew time is 0.590s
  Master Down interval is 3.589s
```

Now that we have Arista-B set with a higher priority, let's see how preemption works. Here, I just shut down the SVI for VLAN 99 on Arista-B:

```
Arista-B(config-if-Vl10)#shut
```

In short order, Arista-A becomes the master router:

```
Arista-A(config-if-Vl10)#sho vrrp
Vlan10 - Group 10
  VRF is default
  VRRP Version 2
State is Master
  Virtual IPv4 address is 10.0.10.1
  Virtual MAC address is 0000.5e00.010a
  Mac Address Advertisement interval is 30s
  VRRP Advertisement interval is 1s
  Preemption is enabled
  Preemption delay is 0s
  Preemption reload delay is 0s
  Priority is 100
  Master Router is 10.0.10.2 (local), priority is 100
  Master Advertisement interval is 1s
  Skew time is 0.609s
  Master Down interval is 3.609s
```

Remember when we added Arista-B, it became the backup because Arista-A was already the master, and the priorities were the same? This time, when Arista-B starts advertising itself with a priority of 105, it should immediately take over master duty:

```
Arista-B(config-if-Vl10)#no shut
```

Arista-A gives up its role as master because a better master is available:

```
Arista-A(config-if-Vl10)#sho vrrp
Vlan10 - Group 10
  VRF is default
  VRRP Version 2
```

```
State is Backup
Virtual IPv4 address is 10.0.10.1
Virtual MAC address is 0000.5e00.010a
Mac Address Advertisement interval is 30s
VRRP Advertisement interval is 1s
Preemption is enabled
Preemption delay is 0s
Preemption reload delay is 0s
Priority is 100
Master Router is 10.0.10.3, priority is 105
Master Advertisement interval is 1s
Skew time is 0.609s
Master Down interval is 3.609s
```

Technically, the router preempted when we first changed the priority, but I wanted to show that failing the new master and then bringing it back online would force another preemption.

Most VRRP implementations that I've used allow the VRRP VIP to be the same as the physical interface's IP address on the master router. In fact, the RFC specifically mentions that using the physical IP address of a router will automatically make it the master. As of EOS 4.21.1F, the ability to use the physical interface's IP address as the VIP is not supported and will result in an error:

```
Arista-A(config-if-Vl10)#vrrp 10 ip 10.0.10.2
% Address 10.0.10.2 is already assigned to interface Vlan10
```

One of the interesting features of VRRP that I rarely see used is the ability to serve multiple IP addresses within the group:

```
Arista-B(config-if-Vl10)#vrrp 10 ip 10.0.10.4 secondary
Arista-B(config-if-Vl10)#vrrp 10 ip 10.0.10.5 secondary
```

And here's the result:

```
Arista-B(config-if-Vl10)#sho vrrp
```

```
Vlan10 - Group 10
  VRF is default
  VRRP Version 2
  State is Master
  Virtual IPv4 address is 10.0.10.1
    Secondary Virtual IPv4 address is 10.0.10.4
    Secondary Virtual IPv4 address is 10.0.10.5
  Virtual MAC address is 0000.5e00.010a
  Mac Address Advertisement interval is 30s
  VRRP Advertisement interval is 1s
  Preemption is enabled
  Preemption delay is 0s
  Preemption reload delay is 0s
  Priority is 105
  Master Router is 10.0.10.3 (local), priority is 105
  Master Advertisement interval is 1s
  Skew time is 0.590s
  Master Down interval is 3.589s
```

Of course, we should do any such configuration on all routers within the group.

You can secure VRRP using clear text or encrypted passwords by using the `vrrp group-id authentication` command. Options for encryptions include `text` and `ietf-md5`. Here's how you configure a clear-text password:

```
Arista-A(config-if-Vl10)#vrrp 10 authentication text ILikePie
```

When using a clear-text password, the password is clearly visible, which is likely why they call it clear text. It is also the reason why you shouldn't use it:

```
Arista-A(config-if-Vl10)#sho vrrp int vlan 10 | grep Auth
  Authentication text, string "ILikePie"
```

If you're going to bother using passwords, do yourself a favor and

encrypt them by using the `ietf-md5 key-string string` option:

```
Arista-A(config-if-Vl10)#vrrp 10 authentication ietf-md5 key-
string ILikePie
```

When using MD5 passwords, the password is not shown in the status output, or in the *running-config*:

```
Arista-A(config-if-Vl10)#sho active | grep auth
vrrp 10 authentication ietf-md5 key-string 7 6m3bUcIbCv6TLQ1FqGARXQ==
```

NOTE

Though much of EOS has migrated to using sha512 for password encryption, as of EOS 4.21.1F, VRRP is still limited to MD5.

I'm going to remove that to simplify the configuration for the rest of the chapter:

```
Arista-A(config-if-Vl10)#no vrrp 10 authentication ietf-md5 key-
string
```

The last thing I cover regarding VRRP is the ability to track other interfaces. Even though I think this has more value on WAN routers for which serial links are likely to fail in colorful and interesting ways, the ability to track another interface is always welcome, and Ethernet handoffs have become the norm for many areas, so let's see how it works. We begin with a baseline VRRP configuration. Here's the configuration for Arista-B, which is back to having only one group and a priority of 105:

```
Arista-B(config-if-Vl10)#sho active
```

```
interface Vlan10
  ip address 10.0.10.3/24
  vrrp 10 priority 105
  vrrp 10 ip 10.0.10.1
```

Here's the status for this switch:

```
Arista-B(config-if-Vl10)#sho vrrp
Vlan10 - Group 10
  VRF is default
  VRRP Version 2
State is Master
  Virtual IPv4 address is 10.0.10.1
  Virtual MAC address is 0000.5e00.010a
  Mac Address Advertisement interval is 30s
  VRRP Advertisement interval is 1s
  Preemption is enabled
  Preemption delay is 0s
  Preemption reload delay is 0s
  Priority is 105
Master Router is 10.0.10.3 (local), priority is 105
  Master Advertisement interval is 1s
  Skew time is 0.580s
  Master Down interval is 3.580s
```

The first step in tracking another interface is to create an object to track. Currently, as of EOS version 4.21.1F, the only objects that can be created are *interface line protocol* objects.

NOTE

In the first edition I wrote, “I can see that Arista, being the forward thinkers that they are, have left the doors open for all sorts of objects in the future. I imagine that I’d be able to track IP routes, ARP entries, MAC-table entries, and all sorts of wonderful things, but for now, let’s ignore my rambling daydreams and focus on the type we have available to us.”

I bring this up not to make a statement about Arista, but to point out that I usually suck at predicting the future.

To create an object, you must specify an object name. I'm going to call mine GAD, because I'm the writer, damn it, and I like seeing my initials in print if you haven't guessed that by now. Also, it's late, I'm tired, and thinking of something else seems like real work. After specifying the object name, append the object type to be tracked. Judicious use of tab completion and the question mark will show you that this is the only choice that you can make, unless you'd like to use your own initials, in which case I say, fine be that way:

```
Arista-B(config-if-Vl10)#track GAD interface e11 line-protocol
```

After you create the object, you can reference it by using the **vrrp group-number track object-name** interface command. There are two options: **decrement** and **shutdown**. **decrement** lowers the priority by the specified amount, and **shutdown** disables the VRRP group entirely. For this example, I decrement the priority by 10:

```
Arista-B(config)#int vlan 10
Arista-B(config-if-Vl10)#vrrp 10 track GAD decrement 10
```

With this track configured, should interface e11 go down, the priority for VRRP group 99 will drop 10 points, from 105 down to 95, which would make it five less than Arista-A's default priority of 100. Let's shut down interface e11 and see what happens:

```
Arista-B(config-if-Vl10)#int e11
Arista-B(config-if-Et11)#shut
```

With that done, let's see what VRRP looks like on Arista-B:

```
Arista-B(config-if-Et11)#sho vrrp
Vlan10 - Group 10
```

```
VRF is default
VRRP Version 2
State is Backup
Virtual IPv4 address is 10.0.10.1
Virtual MAC address is 0000.5e00.010a
Mac Address Advertisement interval is 30s
VRRP Advertisement interval is 1s
Preemption is enabled
Preemption delay is 0s
Preemption reload delay is 0s
Priority is 95
Master Router is 10.0.10.2, priority is 100
Master Advertisement interval is 1s
Skew time is 0.620s
Master Down interval is 3.620s
```

Works as expected! So, for the most part, we're done!

Miscellaneous VRRP Stuff

VRRP has some cool features, as we've seen, but for some reason, I'm tickled by the idea that we can shut down a VRRP group like we can an interface. To do so, just use the `vrrp group-id shutdown` interface command:

```
Arista-B(config-if-Et11)#int vlan 10
Arista-B(config-if-Vl10)#vrrp 10 shutdown
```

To bring it back up, negate the command:

```
Arista-B(config-if-Vl10)#no vrrp 10 shutdown
```

We've already seen the results of the `show vrrp` command, but with many groups configured, it can become unwieldy. The command can be modified with the `interface` keyword, which is useful, but for my money the `show vrrp brief` command is the go-to command in busy environments:

```
Arista-B(config)#sho vrrp brie
Interface Vrf          Id  Ver Pri Time  State   VrIps
Vl10      default      10  2   95  3620  Backup  10.0.10.1
```

With track objects configured, you can see them by using the `show track` command. This can be very useful, especially when you have an object tracked in more than one place:

```
Arista-B(config)#sho track
Tracked object GAD is down
  Interface Ethernet11 line-protocol
    6 change, last change time was 0:05:54 ago
  Tracked by:
    Vlan10 vrrp v4 instance (10)
```

This command also has a brief modifier, but be careful because you need to spell out the word *brie*, or else the parser thinks you're specifying the name of a track object:

```
Arista-B(config)#sho track brie
% Tracked object brie does not exist
Arista-B(config)#sho track brief
Tracked object GAD is down
```

VARP

Virtual ARP, or *Layer 3 Anycast Gateway*, is a feature that's so simple, I'm amazed that no one has ever thought of it before. Put simply, multiple devices are configured to respond to ARP and Gratuitous ARP (GARP) requests for a shared virtual IP address using a shared virtual MAC address. Seriously, that's all it does. So, what's so great about that?

First, by the very nature of the fact that both switches in a pair respond

to ARP requests for the same IP address, this means that some devices will learn the MAC address from one switch, whereas other devices will learn from the other switch (assuming a pair). This also means that both switches can actively receive packets destined for the IP address. This active-active behavior is something that neither VRRP nor HSRP can replicate. Not only that, but VARP is completely standards based.

Cisco's Gateway Load Balancing Protocol (GLBP) offers ARP load balancing, which is kind of similar at a high level, but much more complicated to configure and understand, which brings me to the second benefit. VARP takes just two lines of configuration on each switch for a single IP address and one additional line for each additional IP to be shared. There are no options, no timers, and no additional features to worry about. There are also no protocols involved. It just works.

Third, because packets that are received on a VARP address are forwarded to the next-hop destination on the same switch on which they were received (assuming IP routing is enabled and properly configured), these packets never need to traverse the peer link, thus reducing traffic latency, not to mention load on switches that don't really need to carry the traffic in the first place.

This is accomplished by configuring a virtual MAC address globally on each switch. The switches will respond to ARP requests for each of the IP addresses to be shared with this configured MAC address, regardless of the VLAN or interface on which the request is heard.

Let's take a look at how we might use VARP. First, let's take a look at

VRRP in use on a sample network. In [Figure 19-3](#), there are four servers on VLAN 10, all configured to use the VRRP VIP of 10.0.10.1 as their default gateways. When they communicate with the server at 10.0.20.20, the packets all go to the active router, which is Arista-A, where they are forwarded from the SVI on VLAN 20. Because Multichassis Link Aggregation (MLAG) forwards packets from the switch on which they were received, all of these packets will be forwarded from Arista-A interface to the server at 10.0.20.20.

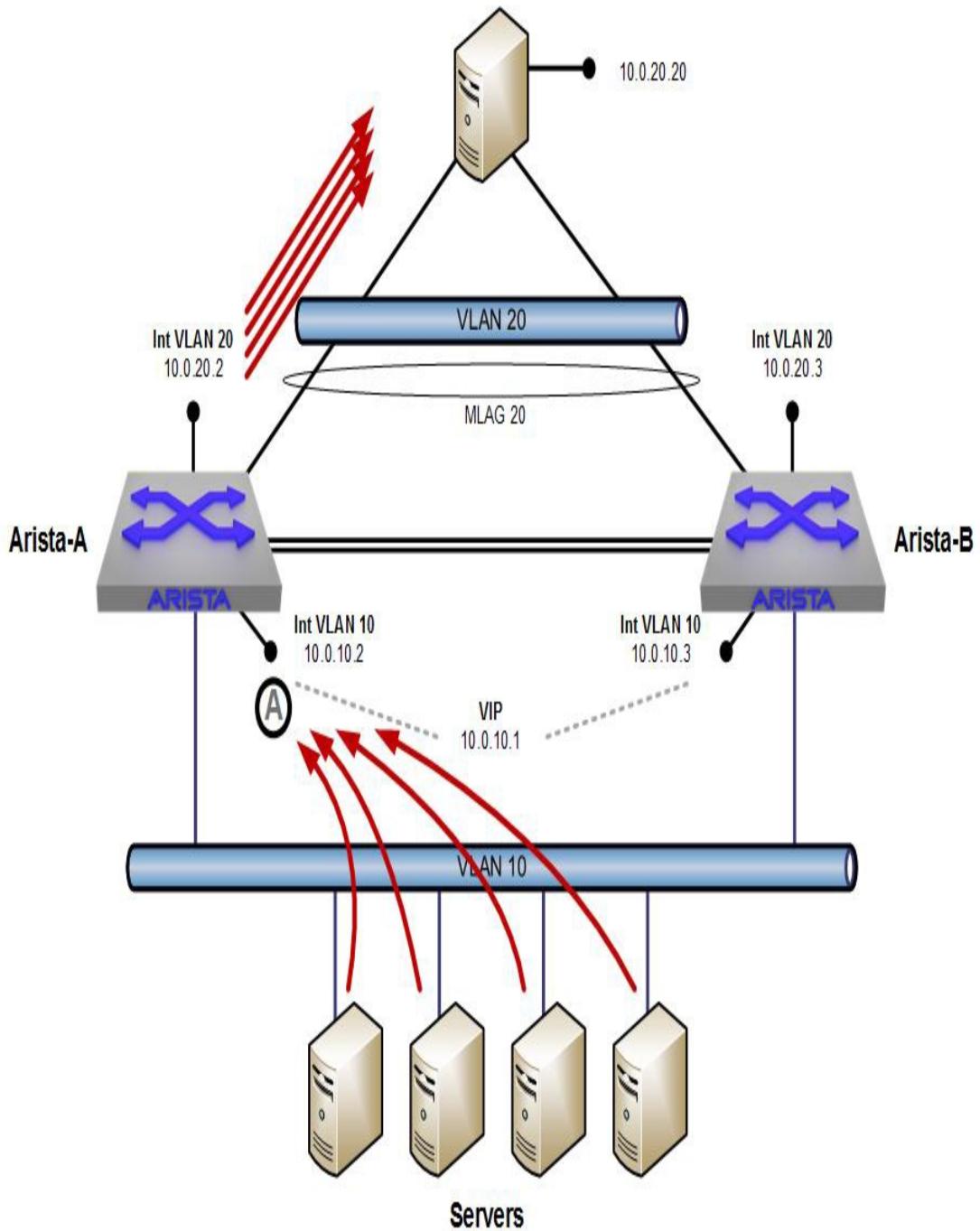


Figure 19-3. Packet flow using VRRP

Now, let's compare that flow with the traffic flow for the same network using VARP. In Figure 19-4, because both switches have responded to the ARP requests for the default gateway of 10.10.10.100, both switches receive packets from a random sampling of the servers that

sent the ARP requests.

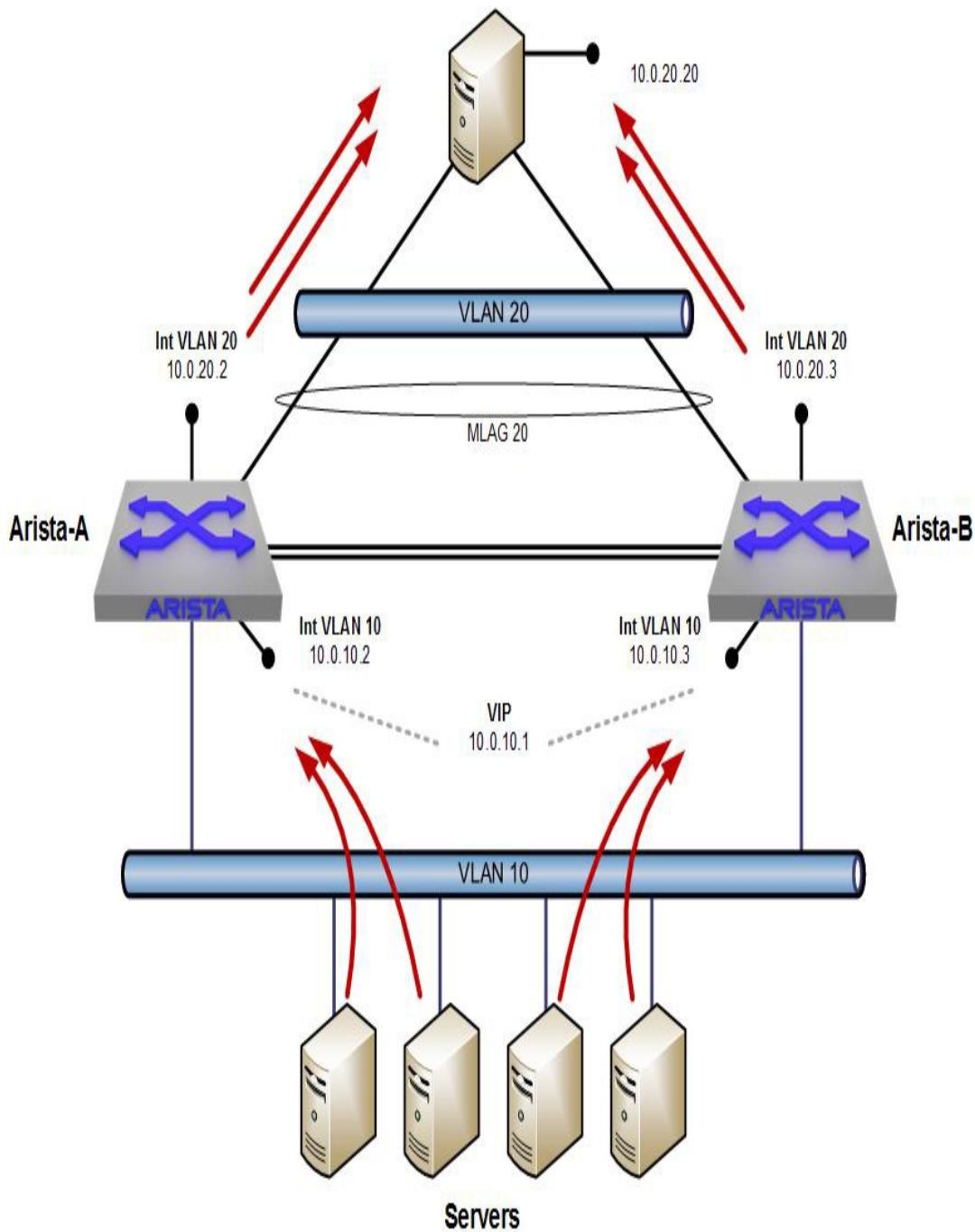


Figure 19-4. A VARP-enabled network

NOTE

Though Figure 19-4 shows a nice, even distribution of servers, this is due to my neurosis that prevents me from crossing lines in drawings unless absolutely necessary. VARP does not weigh, distribute, or otherwise manage load in any way. The servers just believe whatever ARP response they received first, even if more than one gateway responds (see RFC 1027 if you don't believe me).

Because MLAG forwards packets out the same switch wherever possible, the packets are then distributed over both switches, thus better utilizing the switches, the uplinks, and the network in general.

One thing to remember about VARP is that traffic is *never* sourced from the virtual address (with the exception of gratuitous ARPs). This means that the VIP cannot be used in routing, nor can it be used as a source address for protocols or features that support changing the source interface or address.

Configuring VARP

If you're lucky enough to have a lab full of Arista switches, you should clear all the VRRP stuff that we worked on earlier in this chapter. The configuration for the VLAN 10 SVI on Arista-A should look like this to start:

```
Arista-A(config-if-Vl10)#no vrrp 10
Arista-A(config-if-Vl10)#sho active
  interface Vlan10
    ip address 10.0.10.2/24
```

And here's Arista-B's fresh VLAN 10 SVI configuration:

```
Arista-B(config-if-Vl10)#no vrrp 10
Arista-B(config-if-Vl10)#sho active
```

```
interface Vlan10
  ip address 10.0.10.3/24
```

The first step is to configure the MAC address that our MAC address that our VIPs will use. This MAC will be used for every VIP we create on every VLAN, so make it count. I've blatantly stolen this MAC address from the Arista article on VARP because I was too lazy to look up the rules for making up MAC addresses.

NOTE

What can I say? Writing a book involves looking up a lot of things, and I just couldn't bear to look up one...more...damn...thing.

The best part about this MAC address is that it's owned by Arista and exists for your use. It does have the locally administered bit set (that thing I was too lazy to look up), so we're good to go:

```
Arista-A(config)#ip virtual-router mac-address 00:1c:73:00:00:99
```

We need to make it the same on both sides:

```
Arista-B(config)#ip virtual-router mac-address 00:1c:73:00:00:99
```

If you don't do that last step, the next step will fail. The next step is to configure a VIP on an interface, and, if it fails, don't say I didn't warn you:

```
Arista-A(config)#int vlan 10
Arista-A(config-if-Vl10)#ip virtual-router address 10.0.10.1
```

WARNING

You cannot configure VARP on routed Ethernet interfaces, including the management interfaces. You can configure it only on VLAN interfaces. While the Arista *Configuration Guide* doesn't specifically say that it won't work on Ethernet interfaces, to be fair, it only mentions VLAN interfaces in the VARP section. If you need first-hop redundancy on physical interfaces, you'll need to use VRRP.

That's it! The switch is now responding to ARP requests for the VIP 10.0.10.1 with the MAC address that we took so much time to research and create. After these two steps are done, you'll be able to ping the VIP:

```
pArista-A#ping 10.0.10.1
PING 10.0.10.1 (10.0.10.1) 72(100) bytes of data.
80 bytes from 10.0.10.1: icmp_seq=1 ttl=64 time=0.081 ms
80 bytes from 10.0.10.1: icmp_seq=2 ttl=64 time=0.021 ms
80 bytes from 10.0.10.1: icmp_seq=3 ttl=64 time=0.020 ms
80 bytes from 10.0.10.1: icmp_seq=4 ttl=64 time=0.020 ms
80 bytes from 10.0.10.1: icmp_seq=5 ttl=64 time=0.020 ms

--- 10.0.10.1 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.020/0.032/0.081/0.024 ms, ipg/ewma 0.092/0.056
ms
```

NOTE

As an odd bit of history, pinging the VIP didn't work on EOS versions 4.8.0 and prior, and let me tell you that the inability to ping the VIP will drive you slowly mad. Now you know how I got this way.

After you add the virtual address to Arista-B, you'll also be able to ping the VARP address from there:

```
Arista-B(config-if-Vl10)#ip virtual-router address 10.0.10.1
Arista-B(config-if-Vl10)#ip virtual-router mac-address
00:1c:73:00:00:99
Arista-B(config)#ping 10.0.10.1
PING 10.0.10.1 (10.0.10.1) 72(100) bytes of data.
80 bytes from 10.0.10.1: icmp_seq=1 ttl=64 time=0.084 ms
80 bytes from 10.0.10.1: icmp_seq=2 ttl=64 time=0.021 ms
80 bytes from 10.0.10.1: icmp_seq=3 ttl=64 time=0.019 ms
80 bytes from 10.0.10.1: icmp_seq=4 ttl=64 time=0.019 ms
80 bytes from 10.0.10.1: icmp_seq=5 ttl=64 time=0.019 ms

--- 10.0.10.1 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 0ms
```

Not only is this all we need for VARP to work, that's about all that can be done. I like this feature because it's simple, it works, it takes advantage of the way existing, open protocols work, and it's easy to configure.

By the way, you can add more than one virtual IP address per VLAN. To do so, add another IP address with the **ip virtual-router address *ip-address*** interface command. Feel free to add as many as you'd like, up to the limit of 500 per VLAN. I'll add only three so you won't need to flip through 10 pages of code. Besides, my editor would just tell me to take them all out anyway:

```
Arista-B(config-if-Vl10)#ip virtual-router address 10.0.10.11
Arista-B(config-if-Vl10)#ip virtual-router address 10.0.10.12
Arista-B(config-if-Vl10)#ip virtual-router address 10.0.10.13
```

To see the status of your handiwork, use the **show ip virtual router** command, which shows you the configured MAC address and all VIPs, along with the real IP address and the interfaces on which they're operating:

```
Arista-B(config-if-Vl10)#sho ip virtual-router
```

```
IP virtual router is configured with MAC address: 001c.7300.0099
IP router is not configured with Mlag peer MAC address
MAC address advertisement interval: 30 seconds
```

```
Protocol: U - Up, D - Down, T - Testing, UN - Unknown
          NP - Not Present, LLD - Lower Layer Down
```

Interface State	Vrf	Virtual IP Address	Protocol
<hr/>			
Vl10 active	default	10.0.10.1	U
Vl10 active	default	10.0.10.11	U
Vl10 active	default	10.0.10.12	U
Vl10 active	default	10.0.10.13	U

If you configure the interfaces with a virtual router address, but don't configure the global virtual router MAC address your virtual router won't work, even though you might think that it should. Here's what it looks like when starting with no VARP configuration:

```
Arista-B(config)#no ip virtual-router mac-address
00:1c:73:00:00:99
```

It looks fine, but the status tells you it's not. If you're not used to configuring VARP, this can be confusing. You just configured the virtual router address, and it's telling you that the virtual router is not configured:

```
Arista-B(config)#sho ip virtual-router
IP virtual router MAC address is not configured
IP router is not configured with Mlag peer MAC address
```

Conclusion

VARP is a very clever solution to an age-old problem. It was one of the things that piqued my interest when visiting Arista the first time because it's so unique. I encourage you to try it out if at all possible and see how you can use it in your environment. And if you decide that it's not for you, you can always use VRRP. One of the things that VARP has going for it above VRRP, though, is the fact that there are no timers involved, so there's no lag during accidental or even intended failovers.

Chapter 20. FlexRoute

Switches started out being strictly Layer 2 (L2) devices until the advent of the Layer 3 (L3) switch, which, if you are as old as me, you might remember were called *brouters* at their inception. Brouter, being a *portmanteau* of the words bridge and router, was not the kind of word that people enjoyed saying and so was mostly lost to the realm of forgotten terms. “Hey boss, we need another brouter!” <shudder>

Although switches have been able to “do Layer 3 stuff” for a very long time now, it was mostly a convenience thing that allowed us to build networks without having to resort to another terrifically named network design called *router on a stick*. What can I tell you—it was the 90s. L3 switches really changed things in the networking world; although they allowed us to route VLANs between one another, they weren’t really *routers*. This lesson was learned by me the hard way when I decided to have a major provider deliver an OC3 link with an Ethernet handoff so that I didn’t need to spend big money on a router. Routers had WAN interfaces and WAN interfaces were expensive, so I outsmarted the whole system. Hah!

Well, as I soon learned, back then routers supported all sorts of things like QoS and traffic-shaping that we needed, whereas Layer 3 switches did not. Although my L3 switch could route, it wasn’t a real router. Remember, too, that routers often did some heavy lifting at the internet edge where Border Gateway Protocol (BGP) was needed. Supporting

full internet routing tables with hundreds of thousands of prefixes was a complete nonstarter for an L3 switch back in the halcyon days of Cisco 3600 routers and 3550 switches. In fact, it was a complete nonstarter for everyone until Arista came along.

Fast-forward to 2016 or so, and Ethernet handoff has become the norm. No one wants to pay for an OC192 10 Gbps SONET setup when 10 Gbps Ethernet has been standard and getting progressively more inexpensive for years. With many vendors now switching via off-the-shelf Application-Specific Integrated Circuit (ASICs), the ability to forward that kind of traffic in an L3 switch was simple. The problem, though, was still that huge routing table. Not only do most switches not have the memory needed to store that many routes, but actually programming the hardware forwarding tables in the ASIC just wasn't possible. That's not what they were designed for! Arista, however, found a way.

How FlexRoute Works

So how does FlexRoute work? I can't tell you that. Seriously, I asked, and I can't tell you that. What I can tell you is that Arista has figured out a way to make merchant silicon do things that it wasn't designed to do, and that is *seriously freaking cool*. What I can tell you is that this is absolutely a hardware thing, so you can't build this in vEOS (Chapter 32). It is also limited to Arista devices with the "R" suffix, such as the 7280R and the 7500R switches. The switches I'm using for my examples are all model DCS-7280SR-48C6-M-F, so they've got the FlexRoute capability along with the extra memory needed to hold all those routes.

Simulating 800,000 Routes

To show FlexRoute in action, I've built a deceptively simple lab, as shown in Figure 20-1.

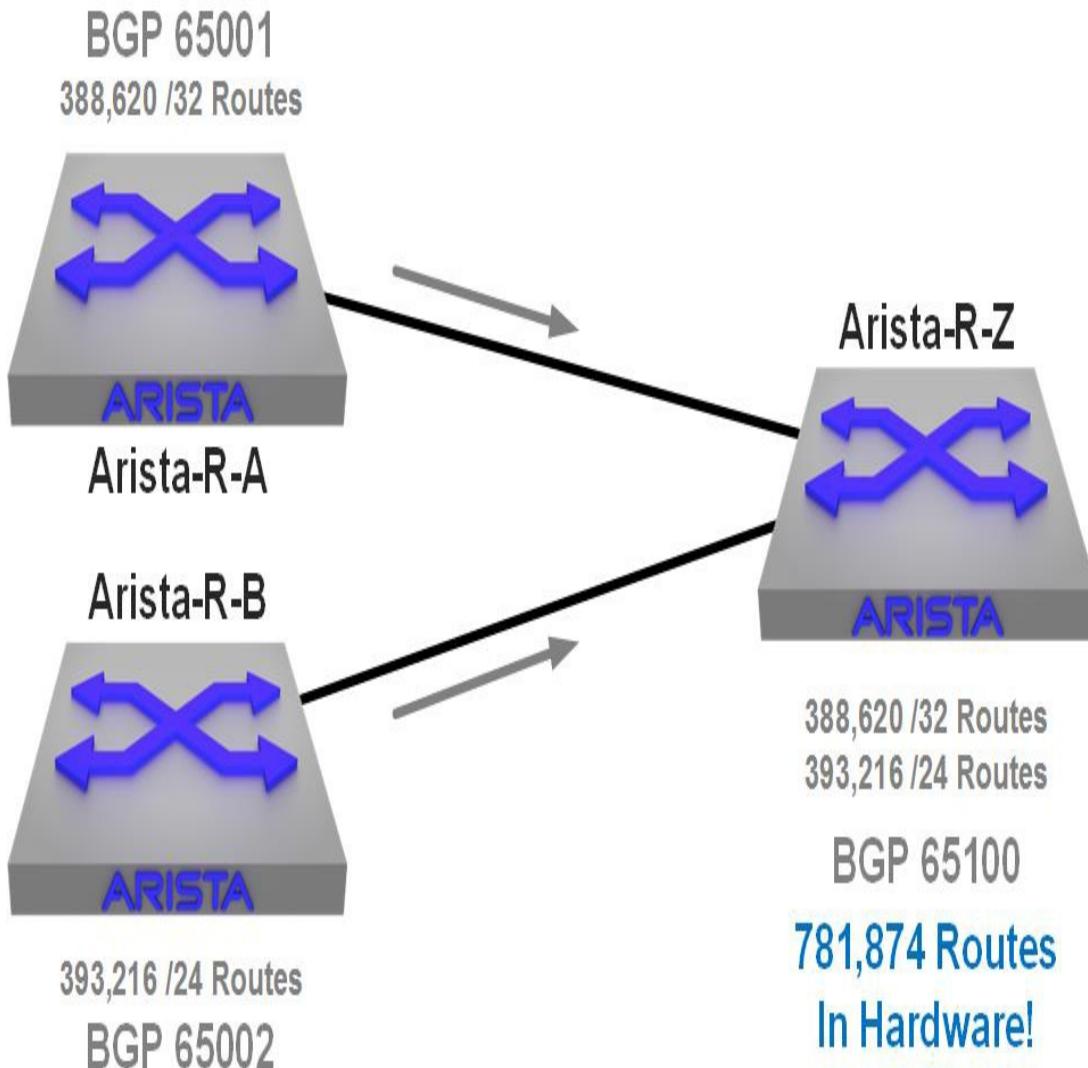


Figure 20-1. Simple FlexRoute lab

Why do I say that it's deceptively simple? Because to make it work, I had to somehow get almost 400,000 routes onto my Arista switches without fancy tools. I didn't have an Ixia, and I like to make my examples repeatable by anyone.

To generate almost 400,000 routes, I wrote a simple eAPI script. Well, two of them, actually: one to create roughly 388,000 /32 routes and another to create around 393,000 /24 routes. Why the strange numbers? For two reasons. First, I went to [CIDR.org](https://www.cidr.org) and determined how big the internet routing table was in terms of number of prefixes. As of today (January 30, 2019) there are 763,024 prefixes in the global internet table, so my goal was to get roughly that many prefixes into one of my 7280Rs via BGP.

Without any sort of sophisticated route injection tool, I decided to resort to Python and eAPI ([Chapter 30](#)) to generate a bunch of routes programmatically. For these to work, eAPI must be enabled to run via localhost. Also, these scripts must be run on the switch itself as written, though that's not a strict requirement if you wanted to change that. Following are the eAPI commands needed on the switch to allow localhost API. Technically the username isn't needed, because I'm connecting via localhost, but I like to boilerplate my configurations, and so it's there regardless of that fact:

```
username Script secret Arista
management api http-commands
  protocol http localhost
  no shutdown
```

Here is the script that I used to generate the /32 routes:

```
#!/usr/bin/python

import jsonrpclib, sys

IP          = "127.0.0.1"
ScriptUser  = "Script"
ScriptPass  = "Arista"
target      = "http://" + ScriptUser + ":" + ScriptPass + "@" + IP + \
```

```

                ":8080/command-api"
switch      = jsonrpclib.Server( target )
allRoutes  = ["enable", "configure"]

print "\n  '.' = 256 routes added.\n  '*' = 65k routes added.\n"

with open("routes.txt", "w") as routeFile:
    routeCounter = 0
    oct2         = 1
    #while oct2 <= 10:
    while oct2 <= 6:
        oct3 = 0
        while oct3 <= 255:
            oct4 = 0
            while oct4 <= 254:
                newRoute =      "ip route 11." + \
                                str(oct2) + "." + \
                                str(oct3) + "." + \
                                str(oct4) + " 255.255.255.255 Null0"
                allRoutes.append(newRoute)
                routeCounter += 1
                oct4 += 1
            sys.stdout.write(".")
            # Comment out next line for testing
            response = switch.runCmds( 1, allRoutes )
            del allRoutes[:]
            allRoutes = ["enable", "configure"]
            oct3 += 1
        sys.stdout.write("*")
        oct2 += 1

print "\n  " + str(routeCounter) + " routes created. \n"

```

And here is the script that I used to generate the /24 routes. This script was run on Arista-R-B:

```

#!/usr/bin/python

import jsonrpclib, sys

IP          = "127.0.0.1"
ScriptUser = "Script"
ScriptPass = "Arista"
target      = "http://" + ScriptUser + ":" + ScriptPass + "@" + IP + \
              ":8080/command-api"

```

```

switch      = jsonrpclib.Server( target )
allRoutes  = ["enable", "configure"]

print "\n  '.' = 256 routes added.\n  '*' = 65k routes added.\n"

with open("routes.txt", "w") as routeFile:
    # Makes 21.0.0.0/24 - 26.255.255.0/24
    oct1  = 21
    routeCounter = 0
    while oct1 <= 26:
        oct2  = 0
        while oct2 <= 255:
            oct3 = 0
            while oct3 <= 255:
                newRoute =      "ip route " + \
                               str(oct1) + "." + \
                               str(oct2) + "." + \
                               str(oct3) + ".0 255.255.255.0 Null0"
                allRoutes.append(newRoute)
                routeCounter += 1
                oct3 += 1
                sys.stdout.write(".")
                # Comment out next line for testing
                response = switch.runCmds( 1, allRoutes )
                del allRoutes[:]
                allRoutes = ["enable", "configure"]
                oct2 += 1
                sys.stdout.write("*")
                oct3 = 0
                oct1 +=1

print "\n\n  " + str(routeCounter) + " routes created. \n"

```

You might notice some weirdness, such as starting at 1 instead of 0 for octets. This was a way for me to tweak the number of routes being generated in a simple way.

I should point out that if you're going to try this, each of these scripts takes approximately *two hours* to run. That's a lot of routes we're putting into the *running-config*, and it just takes time. Here's what the output of one of the scripts looks like:

```
[admin@Arista-R-A ~]$ ./HellaRoutes-32.py
```

'.' = 256 routes added.
'*' = 65k routes added.

Just to reiterate, this takes a long time and will consume a fair bit of the switch's CPU capacity while running. Here's a snippet of `show proc top` from while the script was running:

```
top - 20:48:40 up 3 days, 42 min, 2 users, load average: 1.50, 1.49, 1.24
Tasks: 293 total, 2 running, 291 sleeping, 0 stopped, 0 zombie
%Cpu(s): 29.7 us, 1.7 sy, 0.0 ni, 68.2 id, 0.0 wa, 0.2 hi, 0.1 si, 0.0 st
KiB Mem: 32458980 total, 4711688 used, 27747292 free, 295228 buffers
KiB Swap: 0 total, 0 used, 0 free, 2727296 cached

      PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM     TIME+ COMMAND
 2595 root      20   0 844m 468m 325m S 100.4  1.5 41:24.17
ConfigAgent
 3381 root      20   0 599m 403m 286m S 10.3  1.3 10:04.68 Rib
 2543 root      20   0 687m 366m 238m R  3.6  1.2 19:00.71 Sysdb
 3686 root      20   0 978m 327m 215m S  2.7  1.0 179:36.10 SandFap
```

With my Arista-R-A and Arista-R-B devices fully loaded, I then went to Arista-R-Z to see what BGP had learned. First, here is the memory utilization on Arista-R-Z:

```
Arista-R-Z#sho proc top once | grep Mem
KiB Mem: 32458980 total, 8352096 used, 24106884 free, 275272 buffers
```

8.3 Gb of RAM is being used on this switch. I'm glad I got the switch with the enhanced memory! How about BGP? Let's take a peek at what it sees:

```
Arista-R-Z#sho ip bgp summ
BGP summary information for VRF default
Router identifier 192.168.100.1, local AS number 65100
Neighbor Status Codes: m - Under maintenance
      Neighbor  V  AS      MsgRcvd  MsgSent  InQ  OutQ  Up/Down State
      PfxRcd  PfxAcc
      10.10.1.2  4  65001      1711      30      0      0 20:26:20 Estab
      388621 388621
      10.10.2.2  4  65002      1619      95      0      0 20:26:20 Estab
```

393216 393216

Looks good! So what's the problem? The problem is that those routes are in memory, but that doesn't necessarily mean that they've been programmed into the ASIC's forwarding tables. How can you tell? The easy way is with the `show ip route` command:

```
Arista-R-Z(config)#show ip route

VRF: default
=====
WARNING: Some of the routes are not programmed in hardware, and they are marked with '*'.
=====
[-- route key removed --]

Gateway of last resort:
  S      0.0.0.0/0 is directly connected, Null0

  C      10.0.0.0/24 is directly connected, Management1
  C      10.10.1.0/30 is directly connected, Ethernet1
  C      10.10.2.0/30 is directly connected, Ethernet2
  C      10.10.3.0/30 is directly connected, Ethernet3
  C      10.10.5.0/30 is directly connected, Ethernet5
  C      10.10.7.0/30 is directly connected, Ethernet7
  C      10.10.8.0/30 is directly connected, Ethernet8
  B E    11.1.1.1/32 [200/0] via 10.10.1.2, Ethernet1
*B E   11.1.1.2/32 [200/0] via 10.10.1.2, Ethernet1
  B E   11.1.1.3/32 [200/0] via 10.10.1.2, Ethernet1
  B E   11.1.1.4/32 [200/0] via 10.10.1.2, Ethernet1
  B E   11.1.1.5/32 [200/0] via 10.10.1.2, Ethernet1
  B E   11.1.1.6/32 [200/0] via 10.10.1.2, Ethernet1
  B E   11.1.1.7/32 [200/0] via 10.10.1.2, Ethernet1
  B E   11.1.1.8/32 [200/0] via 10.10.1.2, Ethernet1
  B E   11.1.1.9/32 [200/0] via 10.10.1.2, Ethernet1
  B E   11.1.1.10/32 [200/0] via 10.10.1.2, Ethernet1
  B E   11.1.1.11/32 [200/0] via 10.10.1.2, Ethernet1
  B E   11.1.1.12/32 [200/0] via 10.10.1.2, Ethernet1
  B E   11.1.1.13/32 [200/0] via 10.10.1.2, Ethernet1
*B E   11.1.1.14/32 [200/0] via 10.10.1.2, Ethernet1
*B E   11.1.1.15/32 [200/0] via 10.10.1.2, Ethernet1
*B E   11.1.1.16/32 [200/0] via 10.10.1.2, Ethernet1
*B E   11.1.1.17/32 [200/0] via 10.10.1.2, Ethernet1
*B E   11.1.1.18/32 [200/0] via 10.10.1.2, Ethernet1
```

```
*B E 11.1.1.19/32 [200/0] via 10.10.1.2, Ethernet1
--More--
```

See that warning that I put in bold? That's a problem, and although there aren't that many problems on that screen output, let's use some Linux skills to see how many routes have asterisks next to them. I'm going to issue the `show ip route` command-line interface (CLI) command, pipe that to `grep ^*`, which will search for asterisks as the first character on the line, and then pipe that to `wc -l`, which will report on the number of lines in that output:

```
Arista-R-Z#sho ip route | grep ^* | wc -l
523780
```

Uh-oh! Even though I have almost 800,000 routes in memory, more than 500,000 of them are not programmed into the forwarding tables on the ASIC. That's more than half, and that seems bad. FlexRoute to the rescue.

Configuring and Using FlexRoute

With the problem clearly identified, let's fix it using only three commands on this 7280R switch. But first, a warning.

WARNING

As you are about to see, issuing these commands can have pretty serious consequences, so you should take these steps before you begin receiving routes on the switch. I'm only doing so after we've received almost 800,000 routes to demonstrate the problem and so that you can see the differences with and without FlexRoute involved.

With that out of the way, let's get this switch using FlexRoute. First, we need to tell EOS to deal with routing protocols a little differently than it does by default:

```
Arista-R-Z#conf
Arista-R-Z(config)#service routing protocols model multi-agent
```

WARNING

This command may require a reboot on certain code revisions.

This can cause your switch session to become nonresponsive for a few seconds when there are already hundreds of thousands of routes, so be prepared. Also, all of the routes are removed from the IP Routing table, so that's probably worth a warning, too. That includes the connected routes.

Next, we need to configure the ASIC to optimize its tables (more or less) for what we're trying to accomplish. Normally, on an internet-attached device that's getting full tables from a provider, you would use the following command:

```
Arista-R-Z(config)#ip hardware fib optimize prefixes profile
internet
```

We're not connected to an internet peer, and our tables don't look quite like the real thing because they're all /32s and /24s, so we're going to tweak this a little bit and optimize for only those prefixes:

```
Arista-R-Z(config)#ip hardware fib optimize prefix-length 32 24
! Please restart layer 3 forwarding agent to ensure IPv4 routes are
```

optimized

That seems serious, and it is. Restarting the L3 forwarding agent essentially causes L3 to stop working while the ASIC resets itself to work with FlexRoute. What's more, there is no indication of what command you should actually use to reset that agent. On the 7280Rs, that command is as follows:

```
Arista-R-Z(config)#agent sandL3Unicast terminate
SandL3Unicast was terminated
```

This doesn't seem to have an immediate effect, but it does. Over the next minute or so, routes are being shuffled around by [REDACTED—I said I couldn't tell you that!]. Cool, right? I think so. Kudos to the people who discovered such a cool method for solving this problem. I look forward to reading the patents.

After a couple of minutes, here's what the IP routing table looks like:

```
Arista-R-Z(config)#sho ip route

VRF: default
[-- route key removed --]

Gateway of last resort:
S      0.0.0.0/0 is directly connected, Null0

C      10.0.0.0/24 is directly connected, Management1
C      10.10.1.0/30 is directly connected, Ethernet1
C      10.10.2.0/30 is directly connected, Ethernet2
C      10.10.3.0/30 is directly connected, Ethernet3
C      10.10.5.0/30 is directly connected, Ethernet5
C      10.10.7.0/30 is directly connected, Ethernet7
C      10.10.8.0/30 is directly connected, Ethernet8
B E    11.1.1.1/32 [200/0] via 10.10.1.2, Ethernet1
B E    11.1.1.2/32 [200/0] via 10.10.1.2, Ethernet1
B E    11.1.1.3/32 [200/0] via 10.10.1.2, Ethernet1
B E    11.1.1.4/32 [200/0] via 10.10.1.2, Ethernet1
```

```

B E 11.1.1.5/32 [200/0] via 10.10.1.2, Ethernet1
B E 11.1.1.6/32 [200/0] via 10.10.1.2, Ethernet1
B E 11.1.1.7/32 [200/0] via 10.10.1.2, Ethernet1
B E 11.1.1.8/32 [200/0] via 10.10.1.2, Ethernet1
B E 11.1.1.9/32 [200/0] via 10.10.1.2, Ethernet1
B E 11.1.1.10/32 [200/0] via 10.10.1.2, Ethernet1
B E 11.1.1.11/32 [200/0] via 10.10.1.2, Ethernet1
B E 11.1.1.12/32 [200/0] via 10.10.1.2, Ethernet1
B E 11.1.1.13/32 [200/0] via 10.10.1.2, Ethernet1
--More--

```

Did you notice what's missing? That warning that said, *Some of the routes are not programmed in hardware, and they are marked with ‘*’*, is gone, as are all of those pesky asterisk-marked routes! Being the kind of guy who doesn't believe what he reads in headers, I want to see if there are any routes with asterisks before them:

```

Arista-R-Z(config)#sho ip route | grep ^* | wc -l
0

```

Nice! Before FlexRoute we had 523,780 routes that had not been programmed into hardware, but now they all are! How many routes? Show **IP route summary** to the rescue:

```

Arista-R-Z(config)#sho ip route summary

```

VRF: default	Route Source	Number Of Routes
connected		7
static (persistent)		0
static (non-persistent)		0
VXLAN Control Service		0
static nexthop-group		0
ospf		0
Intra-area: 0 Inter-area: 0 External-1: 0 External-2: 0		
NSSA External-1: 0 NSSA External-2: 0		
ospfv3		0
bgp		781837
External: 781837 Internal: 0		
isis		0

Level-1: 0 Level-2: 0				
rip	0			
internal	25			
attached	2			
aggregate	0			
dynamic policy	0			
Total Routes	781871			
Number of routes per mask-length:				
/0: 1	/8: 3	/24: 393217	/30: 6	/32:
388644				

There are 781,871 routes in total, of which 393,217 are /24s, and 388,644 are /32s. And every single one of them is now programmed into the ASIC's forwarding tables. I don't know about you, but I think that's cool as hell.

Conclusion

FlexRoute is a nifty feature that's available on Arista switches that have an "R" suffix in the model name, such as the 7280Rs used in this chapter. This is a feature with a fairly specialized use case, but it allows us to use our favorite L3 switches and switch operating system for that use. Although that's a bit like marketing, consider that an Arista 7280R can be a whole lot less expensive than a full-blown internet router from another vendor.

Chapter 21. VXLAN

Virtual eXtensible Local Area Network (VXLAN) is a technology that allows devices to communicate on the same Layer 2 (L2) network, even if they are separated by Layer 3 (L3) boundaries. To overly simplify, it's like tunneling L2 over L3, but it's not that easy, because if they made it simple, we wouldn't have jobs.

Networking people have been taught for decades to cut up L2 broadcast domains as much as possible to limit the potential damage caused by things like *broadcast storms*. Additionally, as data centers grew in size, we began running up against limits in the switch hardware such as the maximum number of MAC addresses supported. To that end, we have insisted that data centers have their own IP space. I made a lot of money back in the 90s moving companies off of massive bridged environments and moving them to more logical, segmented IP solutions. So why the step backward?

Solutions such as vMotion allow virtual machines (VMs) to be moved from host to host while keeping the machine available for use. That's a pretty cool thing to do, but to pull it off, the IP address of the VM needs to stay the same. Having data centers in multiple locations is a good thing for disaster avoidance, so those hosts might be in different physical locations. Thus, we have a need for the same IP space to exist in two different physical locations.

Additionally, some clustering technology requires all of the hosts to be within the same IP space. It's common for companies to want to cluster between multiple buildings for resiliency, which again leads us to the requirement for having the same L2 domain in different buildings.

As a long-time network architect, I have spent a fair amount of time rebelling against this. I have some very smart friends who are VMware and storage guys, and they consider me the reason why they can't get their work done. When the network guys insist on a design that prevents the server/storage/VM guys from getting their solutions implemented, the network guys become the roadblock.

The answer (that guys like me really dislike) is that the network exists to serve the connection needs of the servers. If there were no servers, there would be no networks. Because the network hasn't really evolved to meet the needs of modern virtual computing and modern virtual computing keeps insisting on using L2 networks for its solutions, we are stuck trying to make it all work.

There are a couple of existing technologies out there today that try to solve this problem. Cisco's Overlay Transport Virtualization (OTV) uses a network device to tunnel L2 through L3, but it is designed as a point-to-point solution like a network virtual private network (VPN) might be deployed. Because of this, no additional configuration is required on the servers, hosts, or VMware clusters. OTV is Cisco proprietary and requires either a Cisco Nexus switch (a Virtual Device Context [VDC] can be dedicated to OTV) or a Cisco ASR router to function as the OTV devices.

VXLAN is an open standard that was developed with multiple vendors including Cisco, VMware, Arista, and others. Ken Duda, one of Arista's founders, had a major part in its creation.

VXLAN, like most things in IT, is rife with new terms, acronyms, and initialisms, so let's define those before we move on.

VXLAN Overlay Network (Segment)

VXLAN provides an L2 overlay network that exists on top of an L3 network. Yes, that would be L2 tunneled over L3, which is over L2. Each one of the L2 networks is called a *VXLAN Overlay Network* or *VXLAN Segment*. These segments are discrete (there can be many) and are identified by their VXLAN Network Identifiers.

VXLAN Network Identifier (VNI)

The VNI is the means whereby VXLAN keeps the L2 overlay networks separated. It is also sometimes called the VXLAN Segment-ID. The VNI is akin to a VLAN number, but there can be many more VNIs than there can be VLANs on a traditional switch given that the VNI is a 24-bit value (16,777,216 possibilities), whereas the VLAN ID is only a 12-bit value (4,096).

Only hosts on the same VNI can communicate with one another. Hosts are identified by a combination of the host's MAC address and the VNI. Because of this, the same MAC address can appear on multiple VNIs.

VXLAN Tunnel End Point (VTEP)

VTEPs are the devices that encapsulate and tear down VXLAN packets. VTEPs can be part of a hypervisor, but they can also be in separate networking hardware, in which case they are called a *VXLAN Gateway*.

VXLAN Gateway

A VXLAN Gateway is a device outside of a hypervisor that allows non-VXLAN-compliant devices to communicate over VXLAN Overlay Networks. Another term you might see for this is *Hardware VTEP*.

Two additional terms that are thrown around a lot when talking about VXLAN are *underlay* and *overlay*. When we build a VXLAN environment, we're often overlaying an L2 network on top of an existing L3 network. That L3 network is the underlying architecture or *underlay network*, whereas the VXLAN network on top of it is considered to be the *overlay network*.

VXLAN has many facets including VXLAN-bridging and VXLAN routing. In this chapter, I focus on VXLAN bridging because my goal is for you to understand the technology in an efficient amount of space. I can't cover everything, so I've focused on the basic functionality of VXLAN bridging.

Understanding VXLAN

I like to say that VXLAN is nothing more than a VLAN with an X in the middle. What it does is connect VLANs (technically L2 segments) separated by L3 networks. All of the stuff that we'll talk about is the X—it's the technical details that people like us need to know, but in the end it's connecting two (or more) VLANs together, and after you understand how it does that, you'll see that it will behave like...wait for it...a VLAN.

NOTE

Note that Arista's implementation of VXLAN is hardware based and, as such, might not be supported on every Arista switch or software-only devices (VEOS-lab, etc.).

Imagine a company with two data centers connected via an IP cloud. It doesn't matter what the IP cloud consists of. That's why it's a cloud. As an aside, it amazes me how many people don't know that *cloud* means I don't know (and I don't care) what's in there. Anyway, because of demands outside of the network team's control (*cough* VMware *cough*), the order has been passed down that the servers in one data center must be able to communicate with servers in the other data center as if they were in the same L2 network. You know, like a VPN only without all that encryption and security stuff.

In our imaginary company, DC-1 is built using the network 11.0.0.0/8, and DC-2 is built using the network 12.0.0.0/8. The network that needs to exist simultaneously within both is 10.0.0.0/24 and happens to be contained with VLAN 100 on both sides. Figure 21-1 shows this network.

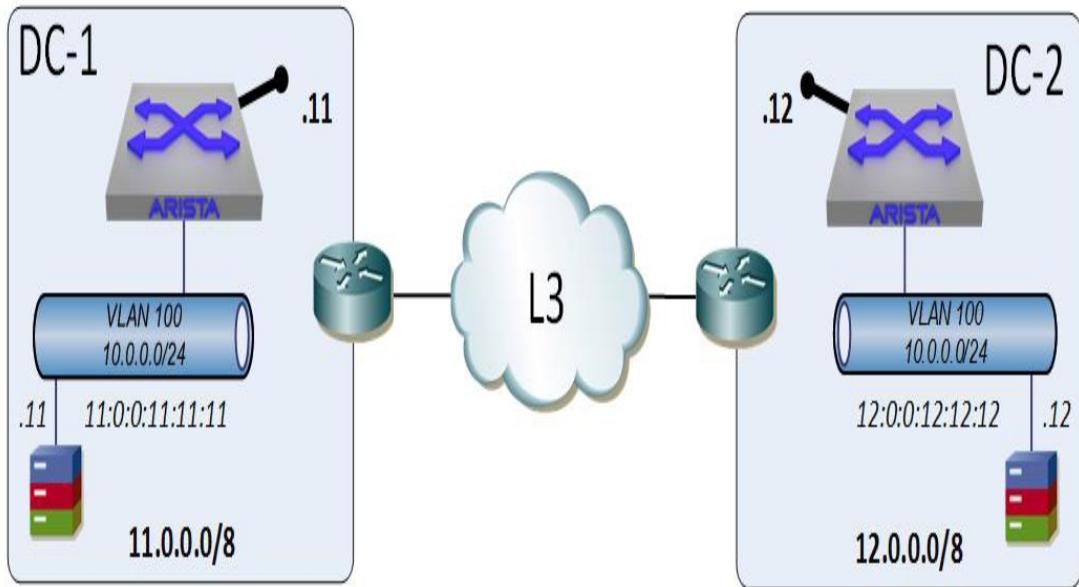


Figure 21-1. Two data centers, both having the same 10.0.0.0/24 networks within them

Because there are multiple L3 networks separating the two VLAN 100 networks, how might we connect them? A VPN between them would work, but this requires specialty hardware due to the encryption that we don't really need. We could build a Generic Routing Encapsulation (GRE) tunnel between them, and that would probably work in this scenario. Heck, there are a bunch of solutions that might work, so let's make it more like the real world in which things are never that simple.

In Figure 21-2 we have the exact same scenario, only now there's a third data center that uses 13.0.0.0/8 while also having the 10.0.0.0/24 network in a VLAN 100. Suddenly, point-to-point solutions don't scale quite as well because we have two destinations that both have the same network (10.0.0.0/24), so we now need to consider things like multipoint solutions.

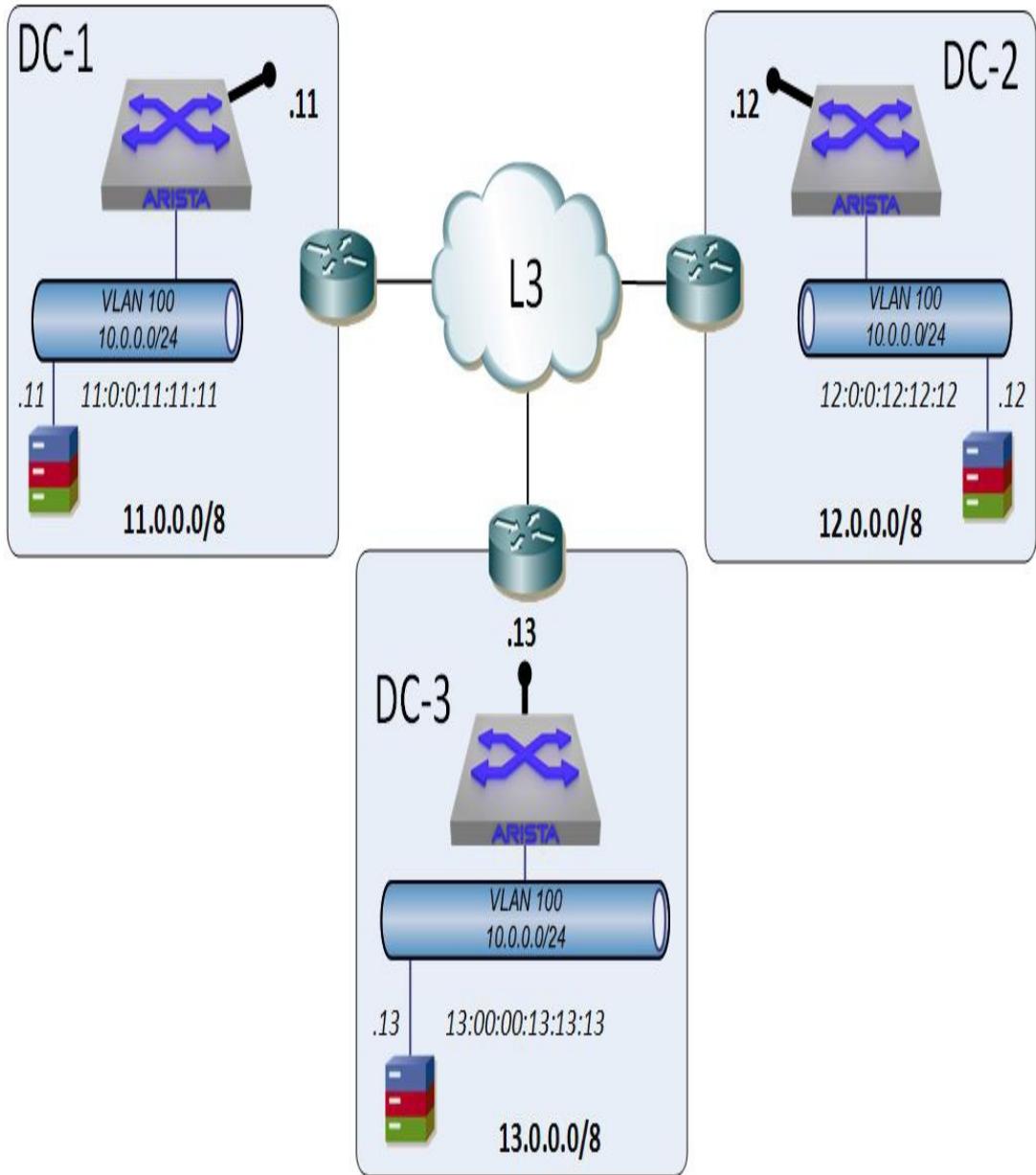


Figure 21-2. Three data centers with the same 10.0.0.0/24 networks

Back when the Cisco Nexus 7000s were new, Cisco released OTV. By putting a Nexus 7000 at the border, you could sort of grab VLAN traffic and tunnel it to another location, thus providing the solution desired (one VLAN in three L3-separated data centers). I looked at OTV when it was young and the problems were many. First, it required Nexus 7000s (it is now supported on other devices) and required a

Virtual Device Context (VDC) to work. What's worse is that it supported only 256 VLANs, so if you needed more, you had to burn another VDC. At the time, I was working for a client that had more than 700 VLANs (and growing) in a multitenancy environment, and when we exceeded the number of VDCs available in the Nexus 7000s (four at the time), the answer from Cisco was to deploy another set of Nexus 7000s just for OTV. My response at the time was simple: nope!

The way that OTV solves the problem at hand is through what Arista calls Data Center Interconnect or DCI. In other words, the VLANs in the data center need to go through a common choke-point where everything is encapsulated, sent to the other side, and then decapsulated. The entire data center is interconnected—hence the name. VXLAN doesn't really work this way (though it can with some clever design work).

In our imaginary company, you no doubt noticed the huge Arista switches in each data center. Note that these are not edge devices, but are within the data centers. These Arista switches are the switches that have VLAN 100 configured on them, so what if we could form a tunnel from them? That's exactly how we will deploy VXLAN, as shown in [Figure 21-3](#).

Instead of an expensive edge device working to capture and encapsulate everything, these devices are encapsulating only what's necessary and only on the configured VLANs. Certainly, OTV works only on the configured VLANs, but I consider it (and most DCI solutions) to be more of a blunt weapon solution, whereas VXLAN is more targeted and refined. There can be problems with that, so don't

think I'm saying that it's all rainbows and unicorns, but we'll get there in due time.

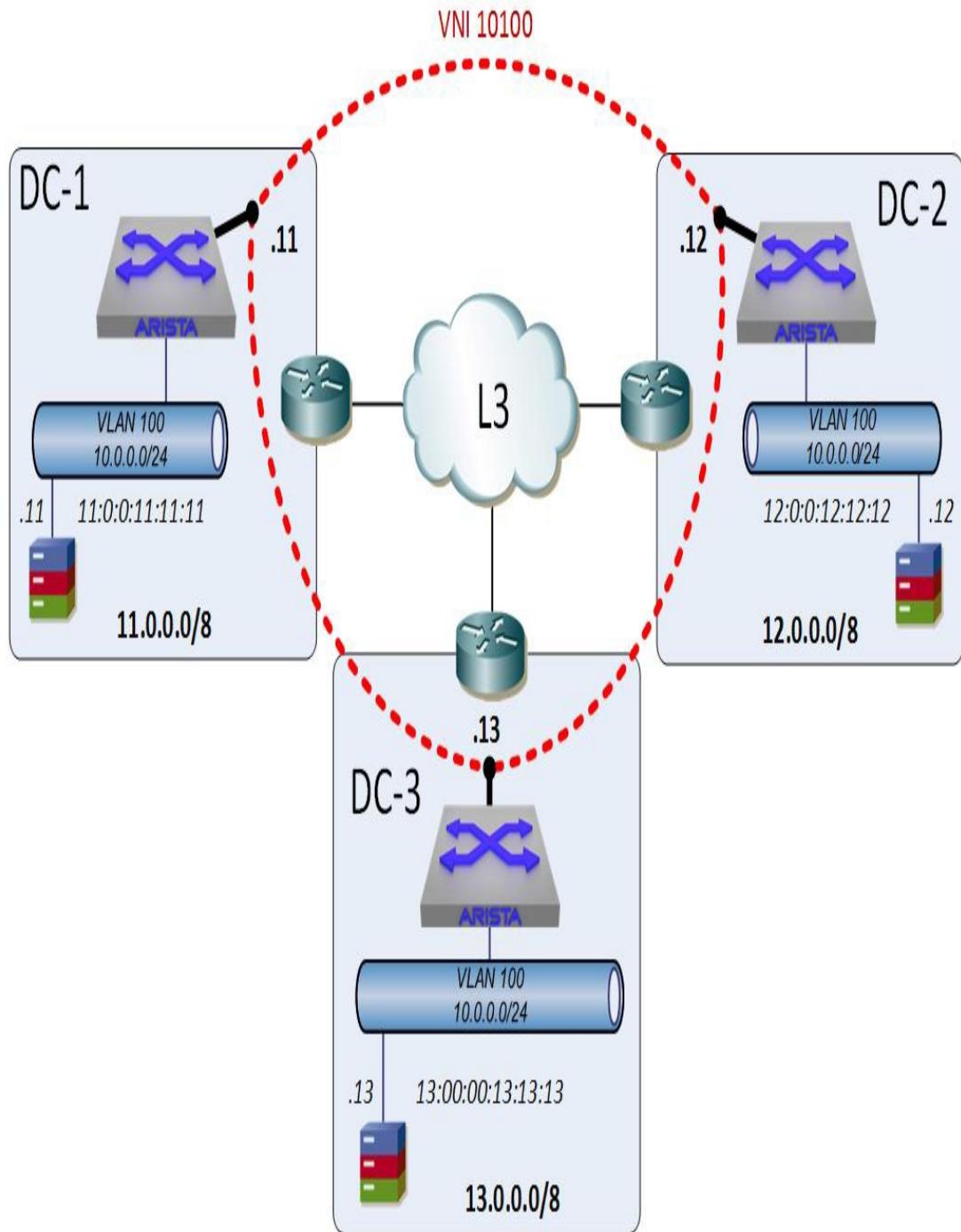


Figure 21-3. VXLAN tunnels

The thing to remember about VXLAN is that it's dynamic. The tunnels (which aren't really tunnels in the strictest sense of the word, as you'll see) come and go as needed, sort of like an on-demand VPN but much simpler and without any perceptible delay.

NOTE

VXLAN is an encapsulation technology. The head-end VTEP takes an L2 frame, wraps it in an L3 packet, and then forwards it to another VTEP. The target VTEP then removes the L3 encapsulation and forwards it as a regular L2 frame. The reason I don't call this a tunnel is because there isn't really a tunnel interface that needs to be up for this to work. Perhaps a better statement would be that VXLAN does not use stateful tunnels like, for example, GRE.

I should point out that VXLAN does not need to be configured on a switch; it could be implemented directly on the VMware ESXi server itself. In my mind, this makes a lot of sense, but in my experience, server teams don't always do network designs very well, and the idea of server teams making hundreds if not thousands of tunnels all across the network in random, unforeseen, and likely unsupported ways keeps me up at night. Plus, this is an Arista book, so I focus on the Arista solution.

Let's review some terms now that we have a pretty picture in [Figure 21-3](#) to look at. Those dotted lines are each a VXLAN Overlay Network or *VXLAN Segment*. The interfaces that they are connected to at each end are *VXLAN Interfaces* on the Arista switches (we see how to configure them in the next section). The Arista switch that contains the VXLAN Interface is the endpoint for the VXLAN Segments and is

therefore called a VXLAN Tunnel End-Point or *VTEP*. To transition to VXLAN-speak, we now have three VXLAN Segments that terminate at three VTEPs.

To understand how all of this really works, I need to shift gears for a moment. Imagine now, a simple switch with a single VLAN. The VLAN has six interfaces, as shown in Figure 21-4.

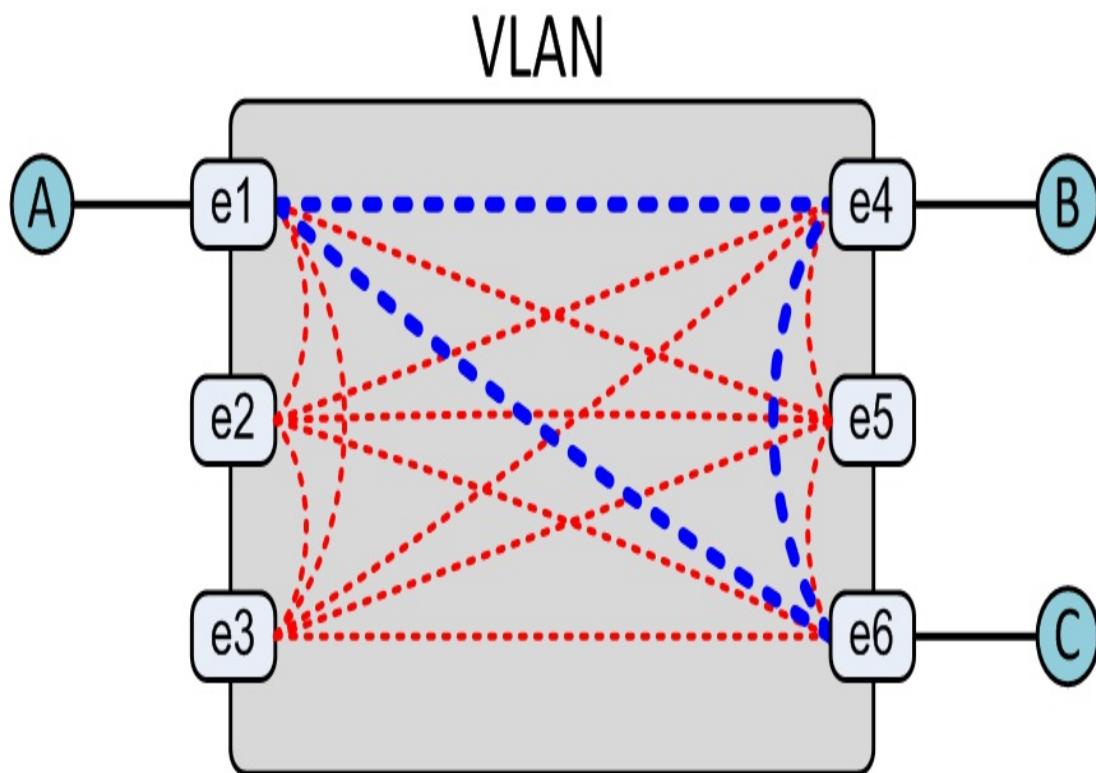


Figure 21-4. Switch VLAN traffic flow

A switch is a switch because there is a dedicated path from every interface to every other interface and each path is only used when needed. This interconnected mesh of paths is called the switch's *fabric*. In Figure 21-4, if host A wants to talk to host B over IP but doesn't know host B's MAC address, what happens? As soon as the destination is determined to be on the same network via a simple subnet mask

XOR operation, a broadcast is sent in the form of an Address Resolution Protocol (ARP) request to every port except the one on which it originated. *Who has the IP address of B?*

This L2 broadcast is received on every connected port, and the host who has that IP address responds via unicast to A. After this happens, the switch knows (because it's been listening to the conversations) where A and B are, and when A sends a packet to B's MAC address, only the proper path is used. When the network is running and stable, the only paths that will be used for A, B, and C to communicate are the bold paths. This is the way switches have worked since there have been switches, and you know what? This is the exact same way that VXLAN works.

Consider now the exact same environment, only instead of a switch fabric and a VLAN connecting hosts, we now have an L3 cloud that's connecting VTEPs, as shown in Figure 21-5.

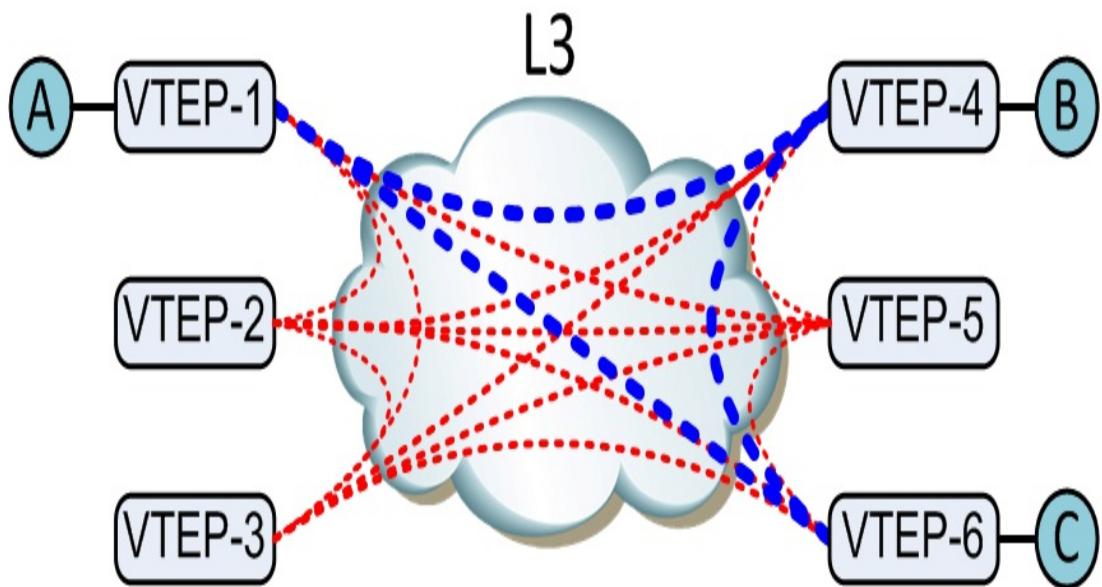


Figure 21-5. VXLAN traffic flow

If host A wants to talk to host B on the same network but doesn't know its MAC address, it sends a broadcast. The same thing happens on the local switch (VTEP-1), but now VXLAN is in the mix, and VXLAN is connecting the VLAN across the L3 cloud.

VXLAN knows the IP addresses of all of the other VTEPs (we see how in a bit) and, just like on a switch sending a broadcast to all interfaces except the one on which it was received, sends the broadcast to all known VTEPs. But how do you send an L2 broadcast across an L3 network? You don't.

What might you use to have the same packet received at multiple different points across multiple L3 networks? Multicast! Only we're not going to do that. Why not? Even though it is supported these days, there are a variety of reasons including things like security, quality of service (QoS) concerns, and some other things that make multicast not the best choice, so we're going to use something else called Head-End Replication (HER).

NOTE

Advanced VXLAN configurations include topics such as VXLAN with Multichassis Link Aggregation (MLAG) and routing over VXLAN, neither of which work with multicast and therefore must use HER or Ethernet VPN (EVPN).

HER simulates a broadcast by replicating the packet being sent at the head end (hence the name). In other words, if host A sends an ARP request broadcast, VTEP-1 will take that frame, encapsulate it into an L3 packet, replicate it, and send a copy to each of the known VTEPs.

The receiving VTEPs will decapsulate the packet and forward the L2 frame as if it were locally originated. When the correct host receives it, it will respond as a unicast, which will then go back across the proper VXLAN Segment (VTEP-4 to VTEP-1) and be decapsulated by VTEP-1. After this happens, each of the VTEPs at either end update their tables so that they don't need to go through the HER broadcast step again. This behavior is the same as a switch updating its mac-address-table, only on a larger scale because it could conceivably be happening on scores of switches miles apart.

HER is done any time a broadcast, an unknown unicast, or a multicast packet needs to traverse the VXLAN network. This traffic is called *BUM traffic*, and no, I didn't make that up.

This is perhaps the toughest thing for people to grasp when it comes to VXLAN, but think of each VTEP as an interface in a VLAN with the VLAN being the L3 cloud and you'll see that it's the same thing. Let's go back to our imagined corporate network and apply what we've learned.

Each VTEP will have an IP address that the other VTEPs will reference, so that IP address must be routable to the other VTEPs. This is the VXLAN interface, and this is where the VXLAN Segments will terminate. In our corporate network the VTEPs are at 11.0.0.11, 12.0.0.12, and 13.0.0.13.

VTEPs know about other VTEPs through something called the *flood-list*. We can manually configure this (which we see in a bit), which is fine for smaller networks, but think of the fact that every VTEP needs

to know the address of every other VTEP and the VLAN/VTI associated with it. If you have 100 VTEPs, the flood-list for every one of them needs to contain the IP address of every other VTEP. That is an administrative nightmare and a recipe for human error, so Arista has come up with a solution using its CloudVision product: CloudVision eXchange, or CVX. CVX dynamically learns the remote VTEP addresses (and associated VXLAN-Enabled L2 segments) and updates the flood-lists for you. I explain how that works later in the chapter, but for right now I'll use a small-scale example and manually configure it.

Another thing that I haven't explained yet is the fact that each VXLAN network is identified by a VNI. Think of the VNI as a VLAN number because that's kind of what it is but with a couple of very important distinctions.

First, whereas a VLAN is identified with a 12-bit number, the VNI is 24-bits, which means that there can be 16,777,216 possibilities. Because a VLAN is mapped to a VNI, VNIs are a great solution for multitenancy because you can map the same VLAN number used by different customers into a different VNI numbers, thus segregating their traffic.

In Figure 21-6, I've added a VNI number and the flood-lists for each VTEP. VLANs are mapped to VNIs, and in this case, I've mapped VLAN 100 to VNI 10100. I often design solutions in which maybe the first two digits are the customer number and the last three are the VLAN, though perhaps four and four digits would scale better. In this example, customer 10's VLAN 100 has been mapped to VNI 10100.

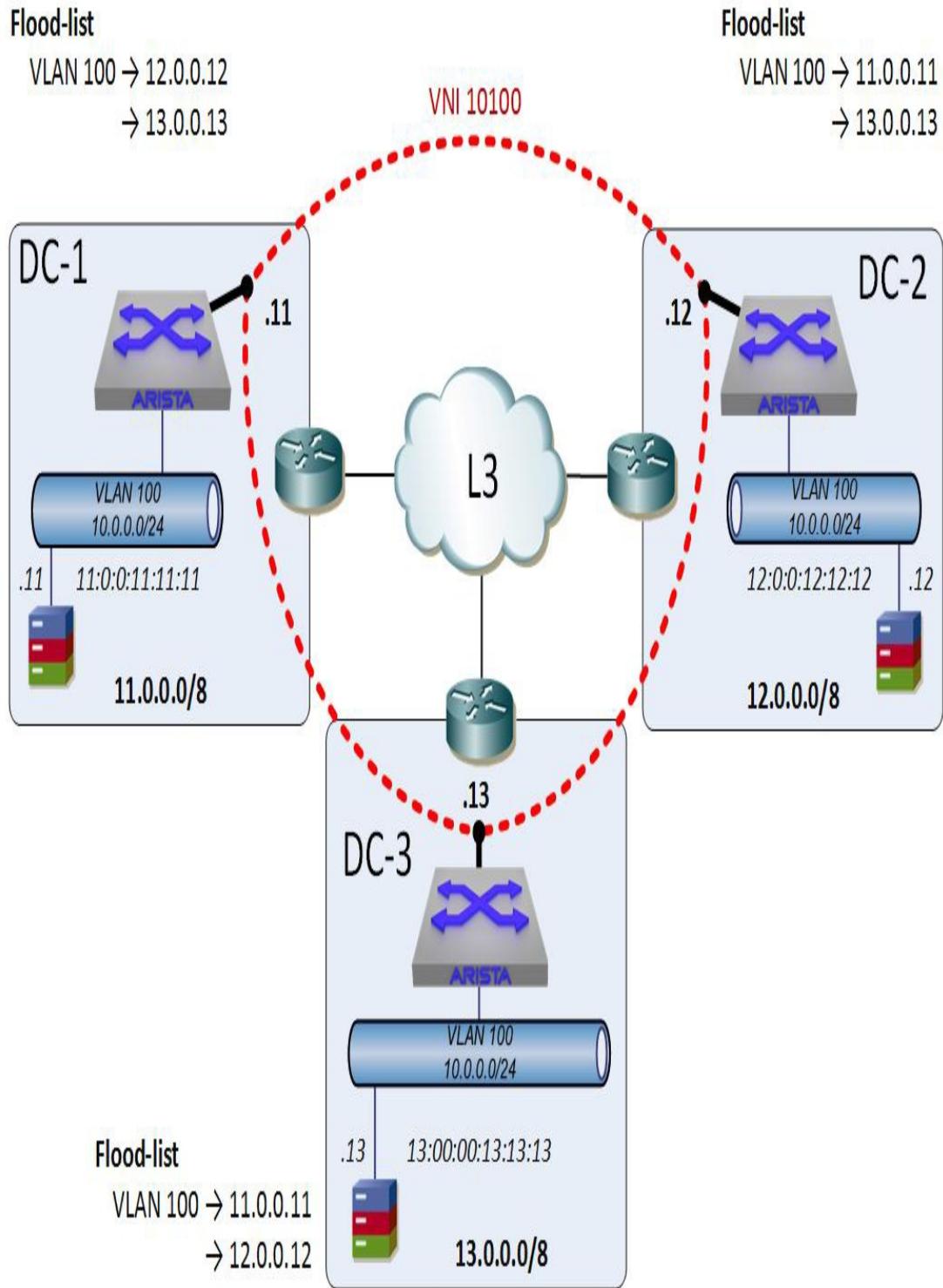


Figure 21-6. VXLAN VNIs and flood-lists

With the flood-lists in place, each of the VTEPs now knows where each of the other VTEPs can be reached. Note that this does not mean

that broadcasts are unnecessary, because a flood-list does not mean that a host behind the VTEP knows how to get to a host behind another VTEP. Think of the flood-list as the way in which you'd create a VLAN and then add interfaces to it in a switch. The flood-list shows each VTEP where the other VTEPs (ports, if you will) are. That's all.

After the VTEPs and flood-lists are in place, the network can begin behaving as we'd expect and as devices start communicating across our VLAN-with-an-X-in-the-middle (VXLAN) network, the VXLAN address-tables begin to populate. These are analogous to a switch's mac-address-table, only instead of a MAC address being behind an interface, the MAC address is behind an IP address (a VTEP), as shown in [Figure 21-7](#).

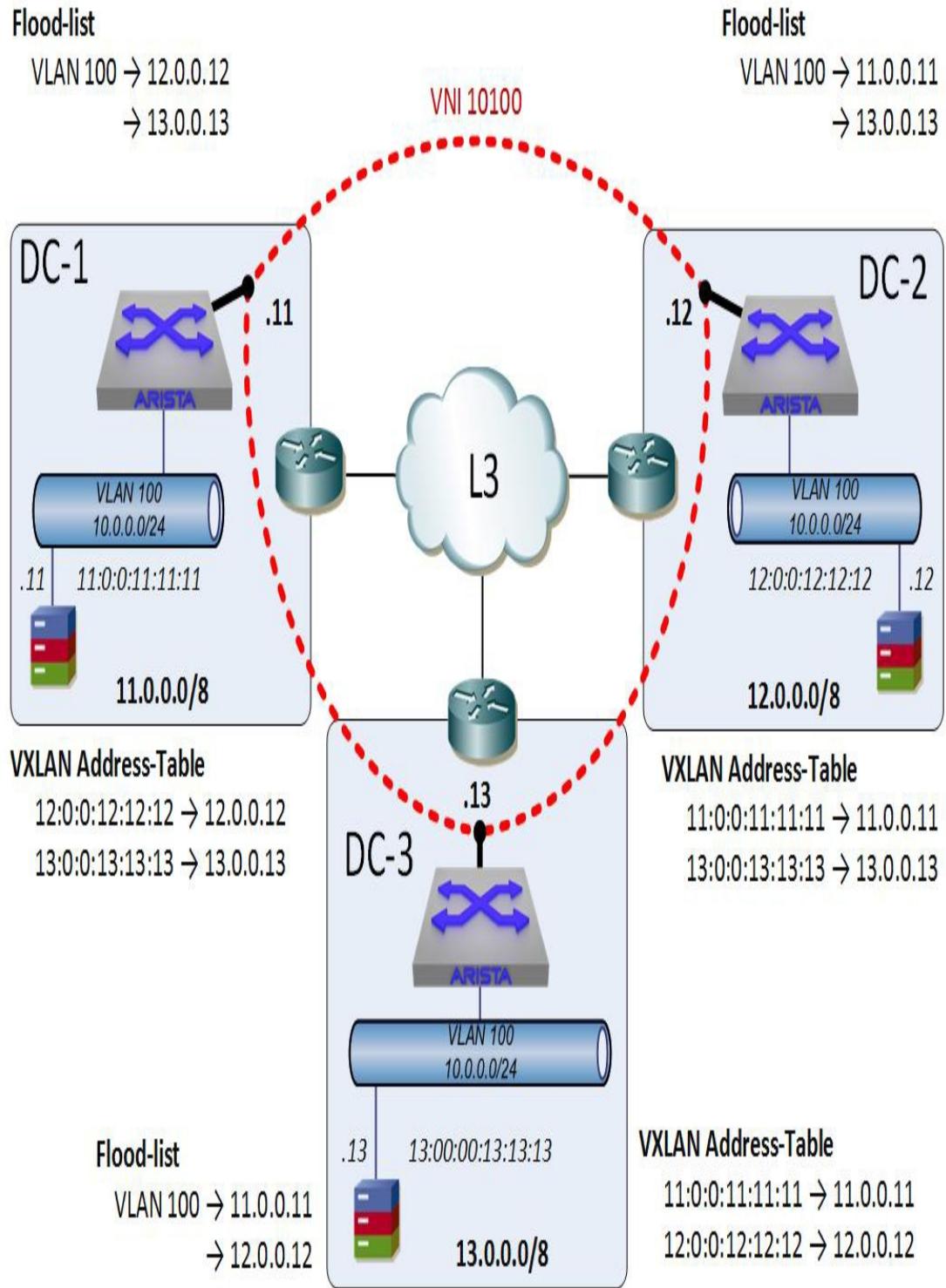


Figure 21-7. VxLAN-address-tables added to our network

With all of this in place, the entire VxLAN environment behaves like a giant switch in which each VTEP is an interface. If the host 10.0.0.11

wants to talk to the host 10.0.0.12, the local switch knows that it's in the VXLAN address-table and encapsulates the frame and forwards it to the VTEP at 12.0.0.12. That VTEP decapsulates the packet and forwards the frame to the proper device.

Configuring VXLAN

Believe it or not, understanding VXLAN is much more difficult than configuring it, at least at the scale we're talking about here. It can be quite complicated in larger environments depending on how you configure the control-plane.

Two of my least-favorite terms in networking are *control-plane* and *data-plane*, but in the interest of not getting a thousand emails from angry nerds (it wouldn't be the first time!) I'll save that rant for another time.

NOTE

I am the king of the angry nerds, so I totally get it. I'm also the king of strong opinions, even if they're not always popular in the industry. Basically, I'm a king with many crowns, including the one for banging angry chapter notes into my laptop while sitting alone in Starbucks.

Here's how the terms are used in VXLAN:

Control-plane

The means whereby VTEPs know or learn about other VTEPs. This is essentially the configuration (be it manual or dynamic) of the

flood-lists, VLAN-VNI mappings, and anything having to do with VTEP learning.

Data-plane

The actual encapsulation/decapsulation of L2 frames into L3 packets and the subsequent forwarding of encapsulated packets to remote VTEPs.

There are three major ways in which a VXLAN control plane is configured, which I describe as manual, CVX, and routing protocol. Before we look at those, let's take a look at the network I'll be using to explain them. To simulate the network I've shown earlier in this chapter, I've created the layout presented in [Figure 21-8](#).

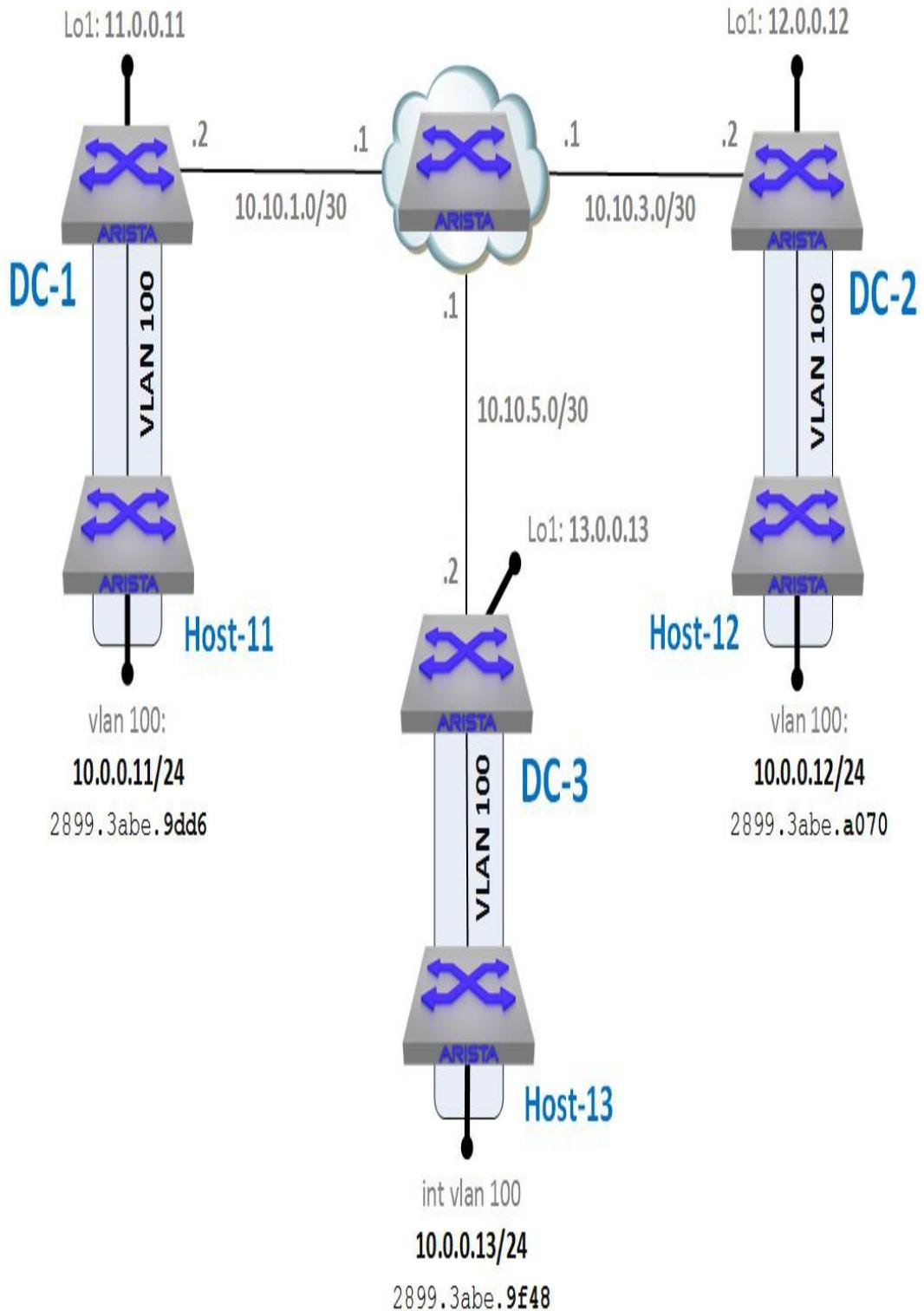


Figure 21-8. VXLAN lab layout

The layout is pretty much the same, with DC1, DC2, and DC3 being

simulated with Arista switches named as such. The hosts in each data center are named Host-11, Host-12, and Host-13 accordingly. Looking at the drawing, you can see that each host and data center contains a VLAN 100, which is local to that data center. Each host has an SVI for VLAN 100 with a 10.0.0.x/24 IP address that associates with the data center in which it is located. Host-11 has the IP address of 10.0.0.11, Host-12 has the IP address of 10.0.0.12, and Host-13 has the IP address of 10.0.0.13. Note that these VLANs are local to each of the data centers, and, as such, they cannot directly communicate with each other because the data centers are separated by L3 routing. That routing happens to be Border Gateway Protocol (BGP) in the lab, but that is essentially irrelevant in these examples (though it can help to explain some of the output we'll see).

Given this layout, trying to ping host 10.0.0.12 or 10.0.0.13 from 10.0.0.11 fails:

```
Host-11#ping 10.0.0.12
PING 10.0.0.12 (10.0.0.12) 72(100) bytes of data.

--- 10.0.0.12 ping statistics ---
5 packets transmitted, 0 received, 100% packet loss, time 4004ms

Host-11#ping 10.0.0.13
PING 10.0.0.13 (10.0.0.13) 72(100) bytes of data.

--- 10.0.0.13 ping statistics ---
5 packets transmitted, 0 received, 100% packet loss, time 4004ms
```

This makes perfect sense due to the fact that the L2 VLANs in each data center are not connected at L2 to the other data centers. We can fix that with VXLAN.

Figure 21-9 presents the same drawing with the VXLAN Segments shown to help you to visualize what we're trying to accomplish.

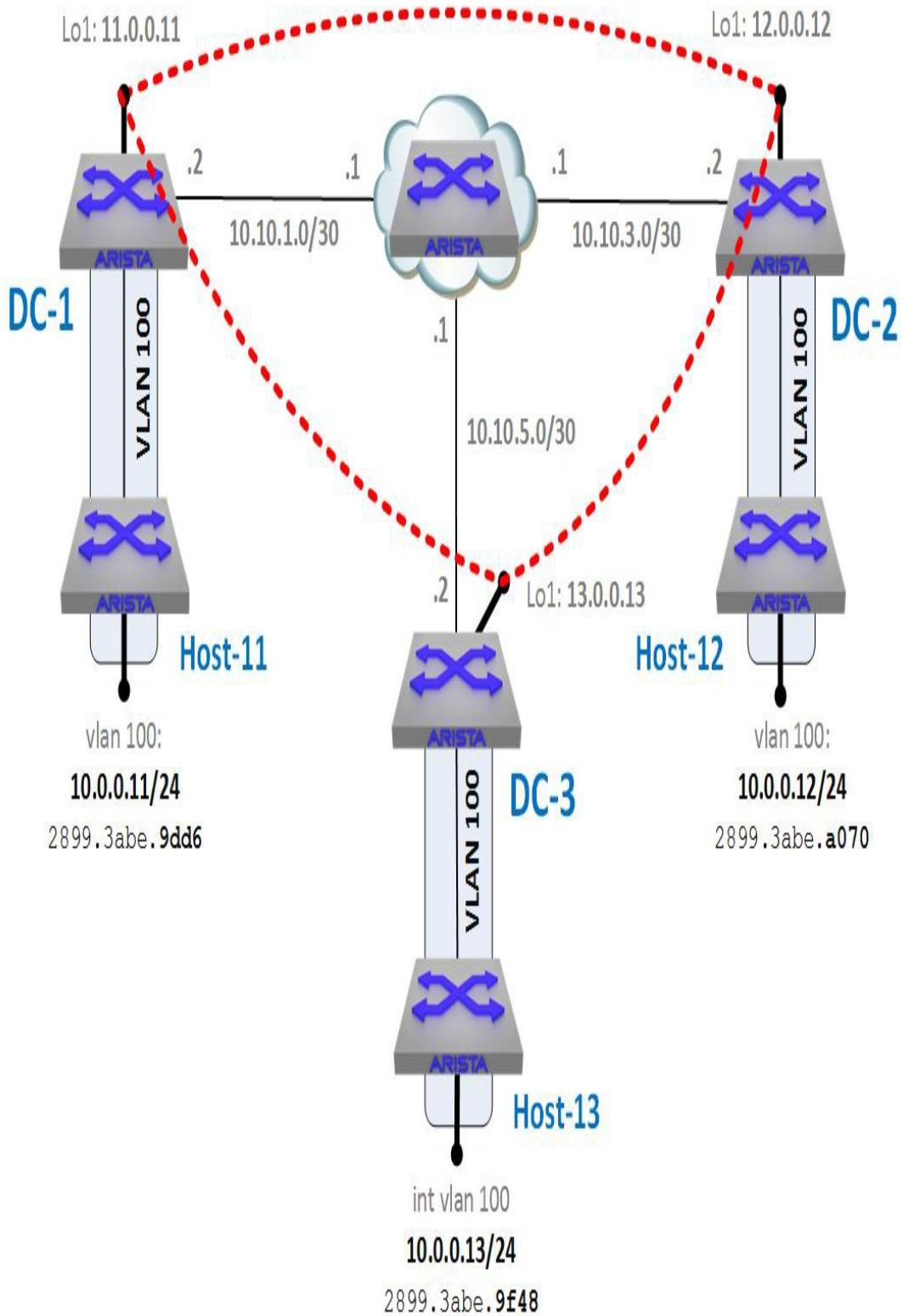


Figure 21-9. VXLAN lab with VXLAN Segments shown

How we accomplish this will vary depending on the control-plane methodology in use, so let's look at them individually.

VXLAN with Manual Control-Plane

Though arguably the simplest of the methods, manually configuring VTEPs and flood-lists does not scale well due mostly to the fact that someone needs to keep track of the VTEPs in the network and manually update all of the other VTEPs. That's not a big deal in our lab where there are only three VTEPs, but in a dynamic or large environment, that task would quickly become unbearable. Still, I like it because it helps to get people to understand how VXLAN works.

First, I need to point out that all IP routing is already in place and that the VLANs and port configurations are already correct, so I do not cover them here (to save space).

To get VXLAN to work, we first must configure a VXLAN interface.

```
DC-1(config)#interface Vxlan1
DC-1(config-if-Vx1)#
```

To make VXLAN as resilient as possible, we use the **source-interface** of Loopback-1 (lo1). In a real network, this would be beneficial with multiple paths being available to the switch. In our lab, that's not the case, but you need to use a source-interface regardless of how your network is built.

```
DC-1(config-if-Vx1)#vxlan source-interface loopback 1
```

Next, we need to map the VLAN to a VNI:

```
DC-1(config-if-Vx1)#vxlan vlan 100 vni 10100
```

The VNI is a number between 1 and 16,777,215, although if you'd prefer a more IP-address-looking format, you can configure EOS to accept dotted-decimal notation by using the **vxlan vni notation dotted** global command:

```
DC-1(config-if-Vx1)#vxlan vni notation dotted
DC-1(config)#
```

With dotted-decimal notation, VNIs use the format of 0.0.1 to 255.255.255. Let's see what happened to our VNI after we changed this:

```
DC-1(config)#int vxlan 1
DC-1(config-if-Vx1)#sho active
interface Vxlan1
    vxlan source-interface Loopback1
    vxlan udp-port 4789
    vxlan vlan 100 vni 0.39.116
```

Cool, huh? Maybe not the way I did it, but if you'd rather see a VNI of 0.10.100, this is how you would go about it:

```
DC-1(config-if-Vx1)#vxlan vlan 100 vni 0.10.100
DC-1(config-if-Vx1)#sho active
interface Vxlan1
    vxlan source-interface Loopback1
    vxlan udp-port 4789
    vxlan vlan 100 vni 0.10.100
```

I'm not a huge fan of that format, so let's change it back.

```
DC-1(config-if-Vx1)#no vxlan vni notation dotted
DC-1(config)#int vxlan 1
DC-1(config-if-Vx1)#sho active
interface Vxlan1
```

```
vxlan source-interface Loopback1
vxlan udp-port 4789
vxlan vlan 100 vni 2660
```

Ugh—let's fix that:

```
DC-1(config-if-Vx1)#vxlan vlan 100 vni 10100
DC-1(config-if-Vx1)#sho active
interface Vxlan1
    vxlan source-interface Loopback1
    vxlan udp-port 4789
    vxlan vlan 100 vni 10100
```

That's better. I hope that I don't need to point out that 10100 and 0.10.100 are not the same value, especially after I just showed that they aren't.

Finally, we need to configure the flood lists. EOS actually allows for per-VLAN flood-lists and a default flood-list. To configure a default flood-list, don't specify a VLAN:

```
DC-1(config-if-Vx1)#vxlan flood vtep 12.0.0.12 13.0.0.13
```

As you can see, I added both remote VTEPs on one line. I can also do it like this:

```
DC-1(config-if-Vx1)#vxlan flood vtep 12.0.0.12
DC-1(config-if-Vx1)#vxlan flood vtep add 13.0.0.13
```

Don't forget the **add** keyword! Otherwise, you'll end up overwriting the first entry:

```
DC-1(config-if-Vx1)#vxlan flood vtep 12.0.0.12
DC-1(config-if-Vx1)#sho active
interface Vxlan1
    vxlan source-interface Loopback1
```

```
vxlan udp-port 4789
vxlan vlan 100 vni 10100
vxlan flood vtep 12.0.0.12
DC-1(config-if-Vx1)#vxlan flood vtep 13.0.0.13
DC-1(config-if-Vx1)#sho active
interface Vxlan1
  vxlan source-interface Loopback1
  vxlan udp-port 4789
  vxlan vlan 100 vni 10100
  vxlan flood vtep 13.0.0.13
```

Let's fix that:

```
DC-1(config-if-Vx1)#vxlan flood vtep 12.0.0.12 13.0.0.13
DC-1(config-if-Vx1)#sho active
interface Vxlan1
  vxlan source-interface Loopback1
  vxlan udp-port 4789
  vxlan vlan 100 vni 10100
  vxlan flood vtep 12.0.0.12 13.0.0.13
```

By the way, you might have noticed the addition of the **vxlan udp-port 4789** command even though we didn't add it. This is a configurable option, but because the default port is 4789, EOS adds the command for you to ensure that it works.

I mentioned that there are two ways to manually configure flood-lists, and we just completed the default flood-list method. To configure a flood-list for a VLAN, simply add the VLAN to the command as follows:

```
DC-1(config-if-Vx1)#vxlan vlan 100 flood vtep 12.0.0.12
13.0.0.13
```

This command works similarly to the default syntax in that you can add VTEPs on a single line or add them like this:

```
DC-1(config-if-Vx1)#vxlan vlan 100 flood vtep 12.0.0.12
DC-1(config-if-Vx1)#vxlan vlan 100 flood vtep add 13.0.0.13
```

The results of either method will now show the following configuration:

```
DC-1(config-if-Vx1)#sho active
interface Vxlan1
    vxlan source-interface Loopback1
    vxlan udp-port 4789
    vxlan vlan 100 vni 10100
    vxlan flood vtep 12.0.0.12 13.0.0.13
    vxlan vlan 100 flood vtep 12.0.0.12 13.0.0.13
```

This is not really something you'd see in the real world (assuming a rational configuration), so let's change the default flood-list to something else:

```
DC-1(config-if-Vx1)#vxlan flood vtep 99.0.0.99
DC-1(config-if-Vx1)#sho active
interface Vxlan1
    vxlan source-interface Loopback1
    vxlan udp-port 4789
    vxlan vlan 100 vni 10100
    vxlan flood vtep 99.0.0.99
    vxlan vlan 100 flood vtep 12.0.0.12 13.0.0.13
```

Why do this? It could be that all of your VLANs are using the same VTEPs, so you don't need a per-VLAN scheme. It could be that all of your VLANs except for one use the same VTEPs, so all other VLANs use the default flood-list, whereas specific VLANs use something else. It's all about options and the ability to consolidate the configuration where possible. To show how this might look, let me add a VLAN-VNI map for VLAN 200:

```
DC-1(config-if-Vx1)#vxlan vlan 200 vni 99200
```

Now I have VLAN 100, which is mapped to VNI 10100, and VLAN 200, which is mapped to VNI 99200. I've set up a specific flood-list for VLAN 100, but all other VLANs that have VNI maps use the default:

```
DC-1(config-if-Vx1)#sho active
interface Vxlan1
  vxlan source-interface Loopback1
  vxlan udp-port 4789
  vxlan vlan 100 vni 10100
  vxlan vlan 200 vni 99200
  vxlan flood vtep 99.0.0.99
  vxlan vlan 100 flood vtep 12.0.0.12 13.0.0.13
```

The way to see the flood-lists is by using the `show vxlan flood vtep` command:

```
DC-1(config-if-Vx1)#sho vxlan flood vtep
  VXLAN Flood VTEP Table
  -----
  VLANS          Ip Address
  -----
  100            12.0.0.12      13.0.0.13
  200 *
  *  All VLANs in the indicated VLAN range list are using the
      default VTEP flood list
```

Removing the VLAN 200 map and the default flood-list, here's the configuration for all three of the data center switches from my lab:

DC-1

```
DC-1#sho run sec vxlan
interface Vxlan1
  vxlan source-interface Loopback1
  vxlan udp-port 4789
  vxlan vlan 100 vni 10100
  vxlan vlan 100 flood vtep 12.0.0.12 13.0.0.13
```

DC2

```
DC-2#sho run sec vxlan
interface Vxlan1
    vxlan source-interface Loopback1
    vxlan udp-port 4789
    vxlan vlan 100 vni 10100
    vxlan vlan 100 flood vtep 11.0.0.11 13.0.0.13
```

DC3

```
DC-3#sho run sec vxlan
interface Vxlan1
    vxlan source-interface Loopback1
    vxlan udp-port 4789
    vxlan vlan 100 vni 10100
    vxlan vlan 100 flood vtep 11.0.0.11 12.0.0.12
```

As a reminder, Figure 21-10 illustrates the layout of the lab.

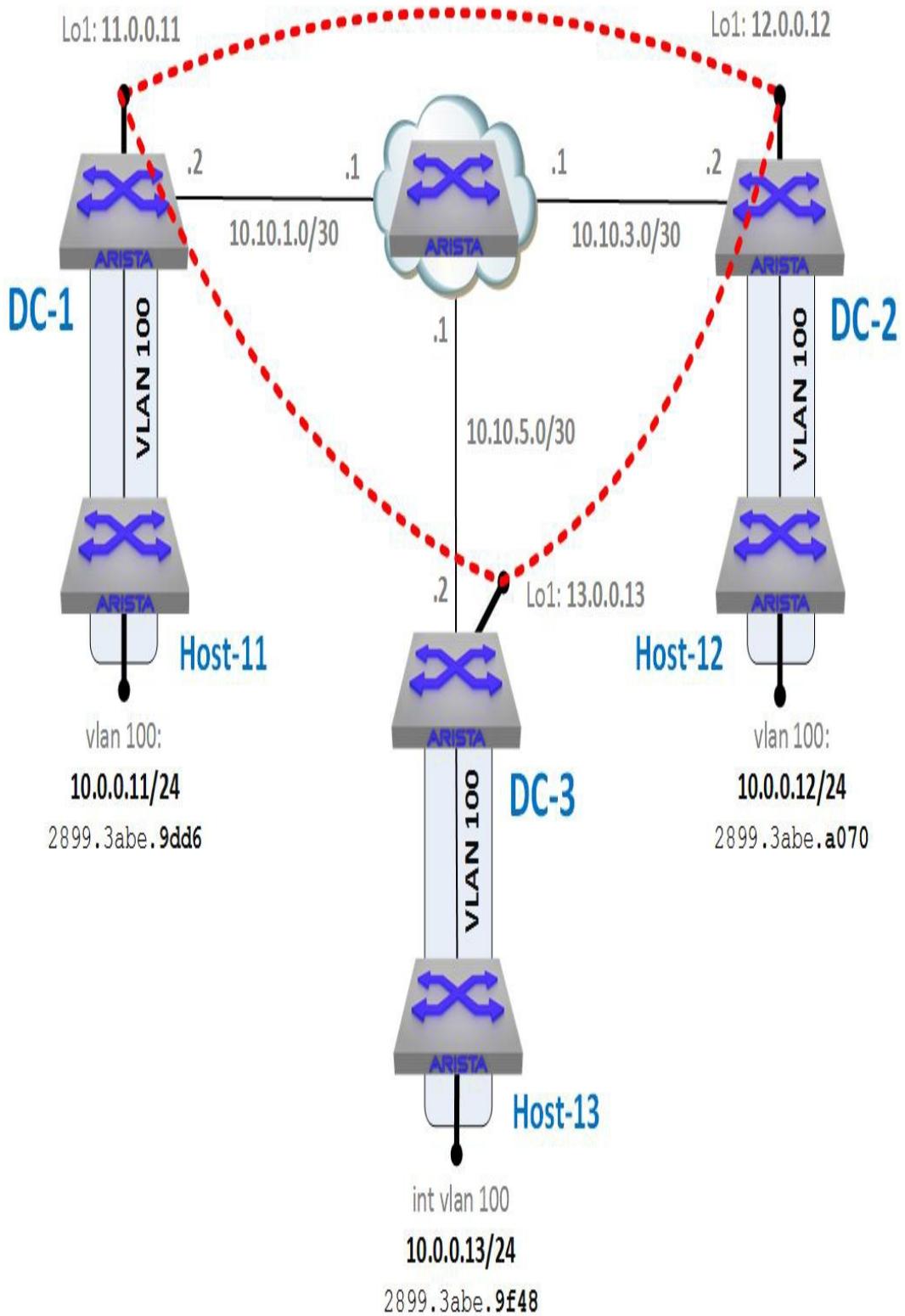


Figure 21-10. VXLAN lab with VXLAN Segments

With the VXLAN interfaces on all three data center switches configured, let's try to ping from Host-11 to Host-12 again:

```
Host-11(vrf:11)#ping 10.0.0.12
PING 10.0.0.12 (10.0.0.12) 72(100) bytes of data.
80 bytes from 10.0.0.12: icmp_seq=1 ttl=64 time=0.263 ms
80 bytes from 10.0.0.12: icmp_seq=2 ttl=64 time=0.113 ms
80 bytes from 10.0.0.12: icmp_seq=3 ttl=64 time=0.104 ms
80 bytes from 10.0.0.12: icmp_seq=4 ttl=64 time=0.109 ms
80 bytes from 10.0.0.12: icmp_seq=5 ttl=64 time=0.109 ms

--- 10.0.0.12 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.104/0.139/0.263/0.063 ms, ipg/ewma 0.236/0.199
ms
```

From DC1, we can see what VTEPs have been used recently with the `show vxlan vtep` command:

```
DC-1#sho vxlan vtep
Remote VTEPS for Vxlan1:
12.0.0.12
Total number of remote VTEPS: 1
```

Nice! How about pinging from Host-11 to Host-13?

```
Host-11(vrf:11)#ping 10.0.0.13
PING 10.0.0.13 (10.0.0.13) 72(100) bytes of data.
80 bytes from 10.0.0.13: icmp_seq=1 ttl=64 time=0.255 ms
80 bytes from 10.0.0.13: icmp_seq=2 ttl=64 time=0.104 ms
80 bytes from 10.0.0.13: icmp_seq=3 ttl=64 time=0.105 ms
80 bytes from 10.0.0.13: icmp_seq=4 ttl=64 time=0.100 ms
80 bytes from 10.0.0.13: icmp_seq=5 ttl=64 time=0.097 ms

--- 10.0.0.13 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.097/0.132/0.255/0.061 ms, ipg/ewma 0.201/0.191
ms
```

Beautiful! Again, from DC1, we can see what VTEPs have been used

recently by using the `show vxlan vtep` command:

```
DC-1#sho vxlan vtep
Remote VTEPs for Vxlan1:
12.0.0.12
13.0.0.13
Total number of remote VTEPs: 2
```

Be aware that this output will be empty unless traffic has been sent to the remote VTEP, and the status times out after about five minutes. This is not a status of the tunnel (because it's not a stateful tunnel); rather (according to the documentation), it *displays the VTEPs that have exchanged data with the configured VTI*. Note also that though I pinged from Host-11, I got the VXLAN VTEP status from DC-1 because Host-11 has no idea that VXLAN even exists.

How about the mac-address-table for VXLAN? You can see this by using the `show vxlan address-table` command. This is similar to the switch's `show mac address-table` command, but instead of a MAC address being found behind an interface, it's found behind an IP address that represents a remote VTEP:

```
DC-1#sho vxlan address-table
Vxlan Mac Address Table
-----
VLAN  Mac Address      Type      Prt  VTEP          Moves  Last Move
-----  -----
 100  2899.3abe.9f48  DYNAMIC  Vx1  13.0.0.13      1      0:00:02 ago
 100  2899.3abe.a070  DYNAMIC  Vx1  12.0.0.12      1      0:03:21 ago
Total Remote Mac Addresses for this criterion: 2
```

If you look at the MAC addresses and compare them to the drawing in [Figure 21-10](#), you'll see how the MAC addresses are found behind the IP address of the VTEP that connects them to the VXLAN Segments.

Another troubleshooting tool at your disposal is the `show vxlan vni` command, which can help to show more information about how VNIs have been mapped:

```
DC-1#sho vxlan vni
VNI to VLAN Mapping for Vxlan1
VNI      VLAN      Source      Interface      802.1Q Tag
-----  -----  -----
10100    100       static      Ethernet48    untagged
```

Note: * indicates a Dynamic VLAN

Although the ability to manually configure VXLAN is tempting for such a small and stable environment as the one in my lab, larger and more complex environments usually rely on some sort of dynamic means for VTEP and flood-list learning. The simplest of those choices, in my opinion, is Arista's own CVX.

VXLAN with CVX

With CVX in the mix, the CVX server (a vEOS instance) handles all of the heavy lifting for things like VTEP discovery, thus alleviating the need for statically configured flood-lists. This employs something called the VXLAN Control Service on CVX, the end result being that the control-plane for VXLAN becomes completely automated.

Consider our original diagram with CVX added, as shown in [Figure 21-11](#). Each of the data center switches subscribes to the CVX server, which then acts as a controller for VXLAN. In this scenario CVX handles all of the control-plane needs for the VTEPs.

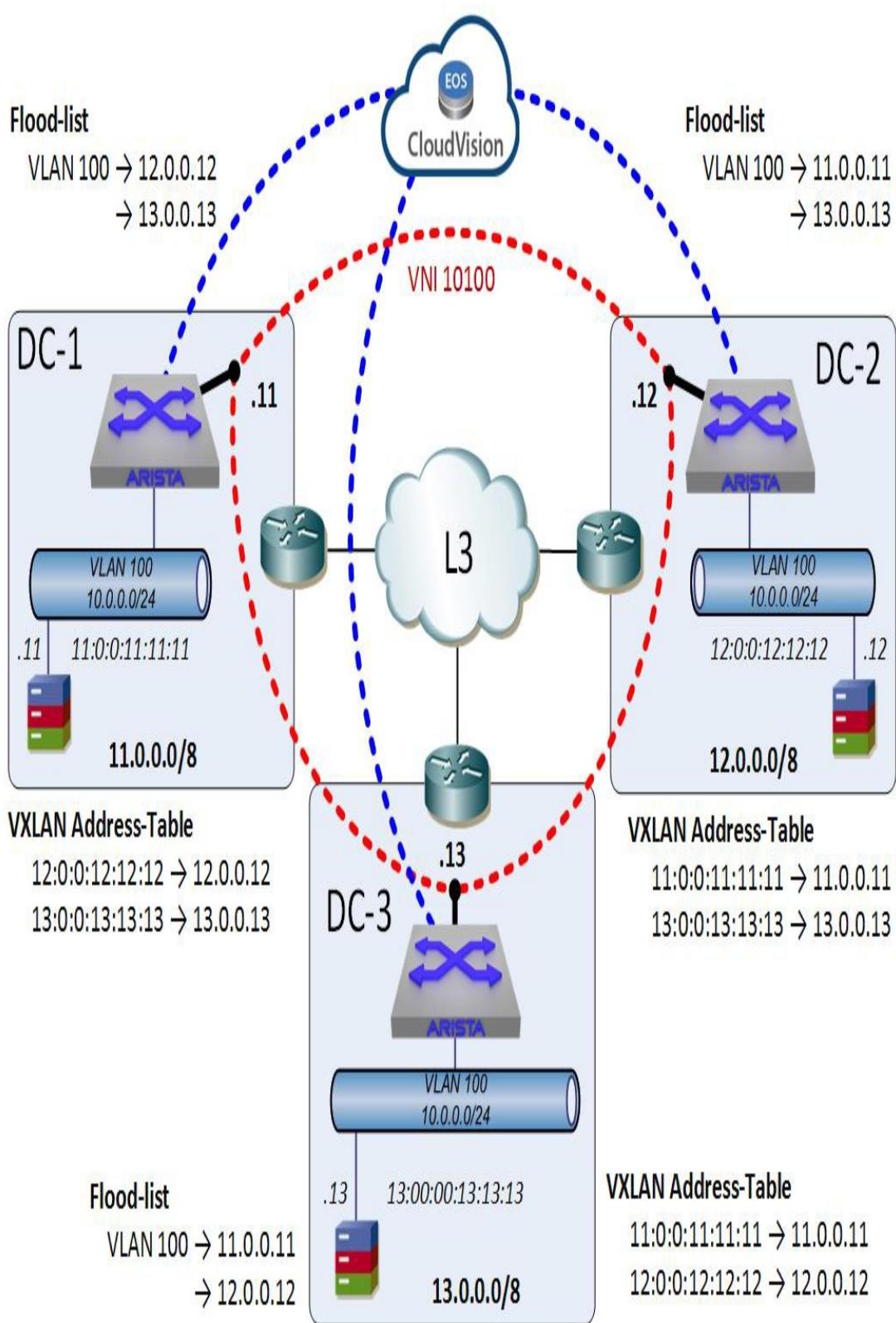


Figure 21-11. VXLAN with CVX

CVX is nothing more than a vEOS VM in this situation, which once again shows the power of vEOS (Chapter 32).

First, here's the configuration of the CVX instance as it relates to our lab:

```
cvx
  no shutdown
  source-interface loopback 1
!
  service vxlan
    no shutdown

hostname CVX

interface Loopback0
  ip address 99.0.0.99/32
```

There is some routing involved to get `Loopback0` into the routing tables of the other switches, but let's not pollute the chapter with that. For my lab network, this is all of the configuration that's necessary (aside from the aforementioned routing) to get CVX working as a VXLAN controller. In fact, it will work without the `source-interface` command and just using the management interface's IP address, but best practice is to have a loopback with multiple paths to it if possible.

On the VTEPs (the data center switches in this lab), the only thing that's required is the following:

```
DC-1(config)#management cvx
DC-1(config-mgmt-cvx)#no shut
DC-1(config-mgmt-cvx)#server host 99.0.0.99
```

We also need to instruct VXLAN that it's running in client mode by

using the `vxlan controller-client` command in the VXLAN interface configuration:

```
DC-1(config)#int vxlan 1
DC-1(config-if-Vx1)#vxlan controller-client
```

That's it! Well, that's it in addition to the normal VXLAN stuff, so let's take a look at the entire thing from the point of view of DC1:

```
DC-1#sho run section vxlan
interface Vxlan1
  vxlan source-interface Loopback1
  vxlan controller-client
  vxlan udp-port 4789
  vxlan vlan 100 vni 10100
```

Note that there is no configuration of a remote VTEP. Here's the configuration for CVX:

```
DC-1#sho run section cvx
management cvx
  no shutdown
  server host 99.0.0.99
```

To see whether the CVX connection is working, use the `show vxlan controller status` command:

```
DC-1#sho vxlan controller status
Controller connection status      : Established
Resync in progress after CVX reboot : No
Purge in progress after CVX reboot : No
Status: Enabled
```

You can also use the `show management cvx status` command for more detail:

```
DC-1#sho management cvx
```

```
CVX Client
Source interface: Inactive (Not configured)
  VRF: default
  Heartbeat interval: 20.0
  Heartbeat timeout: 60.0
  Controller cluster name: default
    Controller status for 99.0.0.99
      Master since 0:15:19 ago
      Connection status: established
      Connection timestamp: 0:15:19 ago
        Out-of-band connection: Not secured
        In-band connection: Not secured (SSL not supported)
      Negotiated version: 2
      Controller UUID: f59be14e-15d9-11e9-bf41-525400230eb3
      Last heartbeat sent: 0:00:19 ago
      Last heartbeat received: 0:00:19 ago
```

The magic word for the connection status is *established*. If it says anything else, it's probably not working, and if it's not working, it's probably something simple because there's not a lot that can go wrong, in my experience, provided you're running all of the VTEPs and the CVX controller on the same (or at least very similar) versions of EOS.

To see the status of the CVX controller in regard to VXLAN, use the `show cvx service Vxlan` command, being careful to note that the V in `Vxlan` is a capital letter:

```
CVX(config-cvx)#sho cvx service Vxlan
Vxlan
  Status: Enabled
  Supported versions: 2

  Switch          Status  Negotiated Version
  -----          -----
  28:99:3a:be:9e:20  Enabled  2
  28:99:3a:be:9f:92  Enabled  2
  28:99:3a:be:a1:04  Enabled  2
```

If this upsets your “but EOS is not case sensitive!” sensibilities, `Vxlan`

is not a keyword, but rather a service name, and the name of the service is *Vxlan*, just like the name of the agent that runs **spanning-tree** is *Stp* with a capital *S*.

There is a pile of other useful commands in CVX, such as **show cvx connections**, which shows the VTEPs in our lab because we're using only the *Vxlan* service in this environment:

```
CVX(config-cvx)#sho cvx connections
Switch 28:99:3a:be:a1:04
  Hostname: DC-3
  State: established
  Connection timestamp: 0:21:07 ago
  Last heartbeat sent: 0:00:07 ago
  Last heartbeat received: 0:00:07 ago
  Out-of-band connection: Not secured
  In-band connection: Not secured (SSL not supported)
Switch 28:99:3a:be:9e:20
  Hostname: DC-2
  State: established
  Connection timestamp: 0:21:07 ago
  Last heartbeat sent: 0:00:07 ago
  Last heartbeat received: 0:00:07 ago
  Out-of-band connection: Not secured
  In-band connection: Not secured (SSL not supported)
Switch 28:99:3a:be:9f:92
  Hostname: Arista
  State: established
  Connection timestamp: 0:21:07 ago
  Last heartbeat sent: 0:00:07 ago
  Last heartbeat received: 0:00:07 ago
  Out-of-band connection: Not secured
  In-band connection: Not secured (SSL not supported)
```

With CVX running, let's take a look at the flood-lists on the data center switches. Here's DC1:

```
DC-1#sho vxlan flood vtep
  VXLAN Flood VTEP Table
-----
```

VLANS	Ip Address		
100	11.0.0.11	12.0.0.12	13.0.0.13

Remember we have not configured any flood-lists, so this list was completely learned via CVX. Here's the output of the same command from DC2:

DC-2#sho vxlan flood vtep			
VXLAN Flood VTEP Table			
VLANS	Ip Address		
100	11.0.0.11	12.0.0.12	13.0.0.13

And, finally, from DC3:

DC-3#sho vxlan flood vtep			
VXLAN Flood VTEP Table			
VLANS	Ip Address		
100	11.0.0.11	12.0.0.12	13.0.0.13

I find it interesting that the flood-list for each of the VTEPs includes itself and upon digging a bit deeper learned that VTEPs in the flood-list containing local addresses are ignored. This makes sense to me because it would allow CVX to send the same flood-list to every VTEP that needs it (in this case, those with the same VNI mapped to VLAN 100).

Another very cool feature of using CVX is that the VXLAN address table is synchronized between all of the VTEPs automatically. To demonstrate this, using a newly rebooted lab as shown earlier, let's ping Host-12 from Host-11. In this scenario DC3 is not involved at all,

but watch what happens:

```
Host-11#ping 10.0.0.12
PING 10.0.0.12 (10.0.0.12) 72(100) bytes of data.
80 bytes from 10.0.0.12: icmp_seq=1 ttl=64 time=0.245 ms
80 bytes from 10.0.0.12: icmp_seq=2 ttl=64 time=0.100 ms
80 bytes from 10.0.0.12: icmp_seq=3 ttl=64 time=0.100 ms
80 bytes from 10.0.0.12: icmp_seq=4 ttl=64 time=0.106 ms
80 bytes from 10.0.0.12: icmp_seq=5 ttl=64 time=0.102 ms

--- 10.0.0.12 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.100/0.130/0.245/0.058 ms, ipg/ewma 0.204/0.186
ms
```

With the ping having been forwarded from Host-11 to DC1, through the VNI to DC2, and then on to Host-12, DC3 is not involved at all, but check out what happens immediately on the DC3 VTEP:

```
DC-3#sho vxlan address-table
Vxlan Mac Address Table
-----
VLAN  Mac Address      Type      Prt  VTEP      Moves  Last Move
-----  -----
 100  2899.3abe.9dd6  RECEIVED  Vx1  11.0.0.11    1    0:00:02 ago
 100  2899.3abe.a070  RECEIVED  Vx1  12.0.0.12    1    0:00:02 ago
Total Remote Mac Addresses for this criterion: 2
```

Even though DC3 had nothing to do with the forwarding of the frame from Host11 (MAC ending in 9dd6) to Host12 (MAC ending in a070), the learned MAC address of both along with the VTEPs where they were discovered have been updated and populated to all of the VTEPs that CVX knows should get the information. That may seem to be a trivial benefit in this small network, but consider the fact that should a host behind DC3's VTEP need to get to either of these hosts, the VTEP already knows about the destination MAC address and the VTEP to

send to without having to send a broadcast (via HER) to every VTEP in the flood-list. This can make the entire environment significantly more efficient, especially in larger environments.

VXLAN with EVPN

Even though CVX is amazingly easy to set up and use (customers have told me that it was the best thing that Arista has ever written), it is an Arista feature and, as such, not supported by other vendors. In a multivendor environment, the solution for discovery of VTEPs is often the open protocol EVPN, which is an address extension of the protocol BGP.

The downside of EVPN is that it's fairly complicated and takes up a lot of configuration space. When I asked around for a deck about EVPN, I was given a 500-plus slide collection, which, by the way, was quite well done. The problem is that 500 slides is more than enough for me to teach a three-day class. After trying in vain for days to try and come up with a way to explain and demonstrate a simple solution for this book, I came to the conclusion that this was not the book for such an example because I could easily consume an additional 30 pages just explaining the way it all works without even getting into VXLAN-specific examples. Although I'm not averse to adding another 30 or more pages, doing so makes the physical book larger and thus makes the price go up. This edition is already larger than the first edition, and so I had to make the difficult decision to not include EVPN.

As a matter of fact, the verbose nature of EVPN has been one of the drivers I've seen that's pushed customers into automation solutions like

CloudVision because it can make the command-line interface (CLI) configuration unwieldy in even smaller environments.

VXLAN with MLAG

VXLAN with MLAG is a bit more complicated than it might seem on the surface, and that's because for VXLAN to be involved, it means that there's an L3 path to the MLAG pair, and this means that we're dealing with a mixed L2/L3 environment. To show how an MLAG pair might be used with VXLAN, I've added a new data center to our lab environment called DC4, as depicted in [Figure 21-12](#).

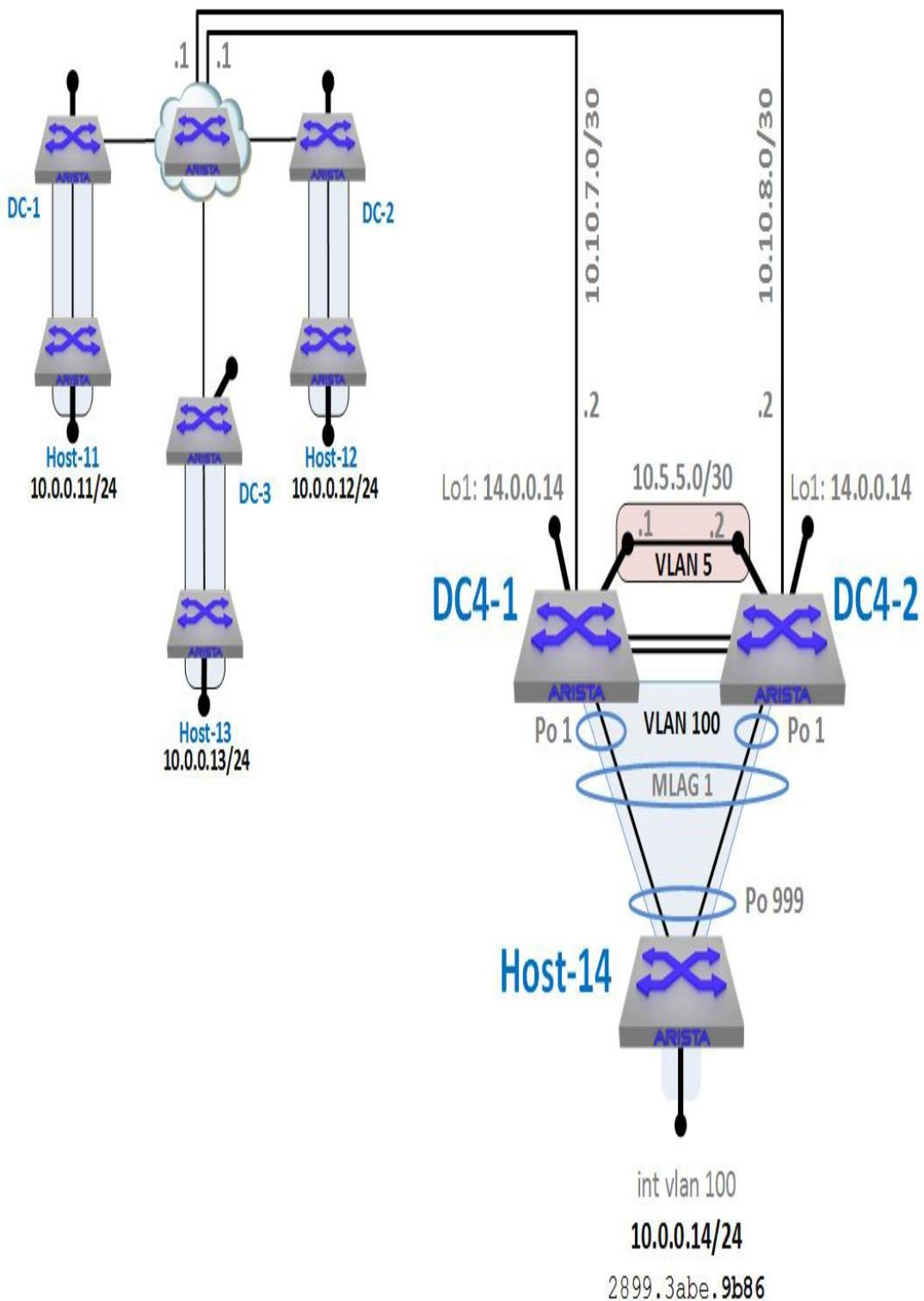


Figure 21-12. MLAG added to our VXLAN lab

Remember that the cloud in the center of our lab is L3, so we need to

have an L3 connection to our new DC4. To do this, we have an L3 link to each of the switches in the MLAG pair. Thus, DC4 is L3 into the cloud, but L2 (using MLAG) down to Host-14.

Because we're using L3, we need an L3 link between the MLAG peers in case one of those L3 links goes down. That interswitch L3 link is VLAN 5 with the IP network of 10.5.5.0/30. More details on configuring such a solution is available in [Chapter 18](#).

The trick for making this design work with VXLAN is that the loopback interfaces on each of the MLAG peers have the same IP address, which in this case is 14.0.0.14. With VXLAN configured to use this interface as the source-interface, and with both switches in the MLAG domain having the same VLAN-VNI mapping and flood-lists, the MLAG pair will behave as a single logical VTEP. Really, that's it! So long as the routing is properly set up and MLAG is properly configured, having the VXLAN information configured the same way on both switches in the domain will make it work. Let's look at the configurations. Here is the relevant configuration for DC4-1:

```
hostname DC4-1
!
no spanning-tree vlan 4094
!
vlan 5,100
!
vlan 4094
  trunk group mlagpeer
!
interface Port-Channel1
  description [ Host-14 MLAG ]
  switchport access vlan 100
  mlag 1
!
interface Port-Channel1000
```

```
        description [ MLAG Peer-Link ]
        switchport mode trunk
        switchport trunk group mlagpeer
!
interface Ethernet31
    description [ BGP Link to Cloud ]
    no switchport
    ip address 10.10.7.2/30
!
interface Ethernet33
    description [ Host-14 ]
    channel-group 1 mode active
!
interface Ethernet47
    description [ MLAG Peer ]
    channel-group 1000 mode active
!
interface Ethernet48
    description [ MLAG Peer ]
    channel-group 1000 mode active
!
interface Loopback1
    ip address 14.0.0.14/8
!
interface Vlan5
    description [ BGP ISL ]
    no autostate
    ip address 10.5.5.1/30
!
interface Vlan4094
    description [ MLAG Link ]
    no autostate
    ip address 10.255.255.1/30
!
interface Vxlan1
    vxlan source-interface Loopback1
    vxlan controller-client
    vxlan udp-port 4789
    vxlan vlan 100 vni 10100
!
ip routing
!
mlag configuration
    domain-id mlag
    local-interface Vlan4094
    peer-address 10.255.255.2
    peer-link Port-Channel1000
```

```

!
router bgp 65007
  maximum-paths 32 ecmp 32
  neighbor 10.5.5.2 remote-as 65008
  neighbor 10.5.5.2 maximum-routes 12000
  neighbor 10.10.7.1 remote-as 65100
  neighbor 10.10.7.1 maximum-routes 12000
  neighbor 10.10.7.5 remote-as 65100
  neighbor 10.10.7.5 maximum-routes 12000
  network 10.1.7.0/24
  network 14.0.0.0/8
!
management cvx
  no shutdown
  server host 99.0.0.99

```

Although there is a lot of added code in relation to MLAG and BGP, if you look at the VXLAN configuration (in bold), you can see that it's really very similar to what we've done in the other data centers. Note the loopback IP address, which is 14.0.0.14/8. You'll see the exact same loopback address in DC14. Here is the relevant configuration for DC14-2, with the VXLAN config again in bold:

```

hostname DC4-2
!
vlan 5,100
!
vlan 4094
  trunk group mlagpeer
!
interface Port-Channel1
  description [ Host-14 MLAG ]
  switchport access vlan 100
  mlag 1
!
interface Port-Channel1000
  description [ MLAG Peer-Link ]
  switchport mode trunk
  switchport trunk group mlagpeer
!
interface Ethernet31
  description [ BGP Link to Cloud ]
  no switchport

```

```
    ip address 10.10.8.2/30
!
interface Ethernet32
    no switchport
    ip address 10.10.8.6/30
!
interface Ethernet33
    description [ Host-14 ]
    channel-group 1 mode active
!
interface Ethernet47
    description [ MLAG Peer ]
    channel-group 1000 mode active
!
interface Ethernet48
    description [ MLAG Peer ]
    channel-group 1000 mode active
!
interface Loopback1
    ip address 14.0.0.14/8
!
interface Vlan5
    description [ BGP ISL ]
    no autostate
    ip address 10.5.5.2/30
!
interface Vlan4094
    description [ MLAG Link ]
    no autostate
    ip address 10.255.255.2/30
!
interface Vxlan1
    vxlan source-interface Loopback1
    vxlan controller-client
    vxlan udp-port 4789
    vxlan vlan 100 vni 10100
!
ip routing
!
mlag configuration
    domain-id mlag
    local-interface Vlan4094
    peer-address 10.255.255.1
    peer-link Port-Channel1000
!
router bgp 65008
    maximum-paths 32 ecmp 32
```

```

neighbor 10.5.5.1 remote-as 65007
neighbor 10.5.5.1 maximum-routes 12000
neighbor 10.10.8.1 remote-as 65100
neighbor 10.10.8.1 maximum-routes 12000
neighbor 10.10.8.5 remote-as 65100
neighbor 10.10.8.5 maximum-routes 12000
network 10.1.8.0/24
network 14.0.0.0/8
!
management cvx
  no shutdown
  server host 99.0.0.99

```

To prove that this is behaving as I've described, I first show the VXLAN flood-lists, which have been populated to each VTEP from CVX given that this is what we're still using for the control-plane.

Here is DC1's VXLAN flood-list:

```

DC-1#sho vxlan flood vtep
  VXLAN Flood VTEP Table
  -----
  VLANS          Ip Address
  -----
  100            11.0.0.11    12.0.0.12    13.0.0.13
                  14.0.0.14

```

We can see here that so far as DC1 (and CVX and every other VTEP in the mix) is concerned, there is a VTEP at IP address 14.0.0.14, which is the IP we configured on **loopback-1** of both of the DC4 MLAG peers. Here's what the same table looks like on DC14-1:

```

DC4-1#sho vxlan flood vtep
  VXLAN Flood VTEP Table
  -----
  VLANS          Ip Address
  -----
  100            11.0.0.11    12.0.0.12    13.0.0.13

```

This shouldn't really be a surprise because CVX is populating these tables, and they should therefore all be the same. As a result of the MLAG pair appearing as a single logical VTEP in the L3 network and as a single logical bridge in the L2 network, [Figure 21-13](#) shows how this design appears logically.

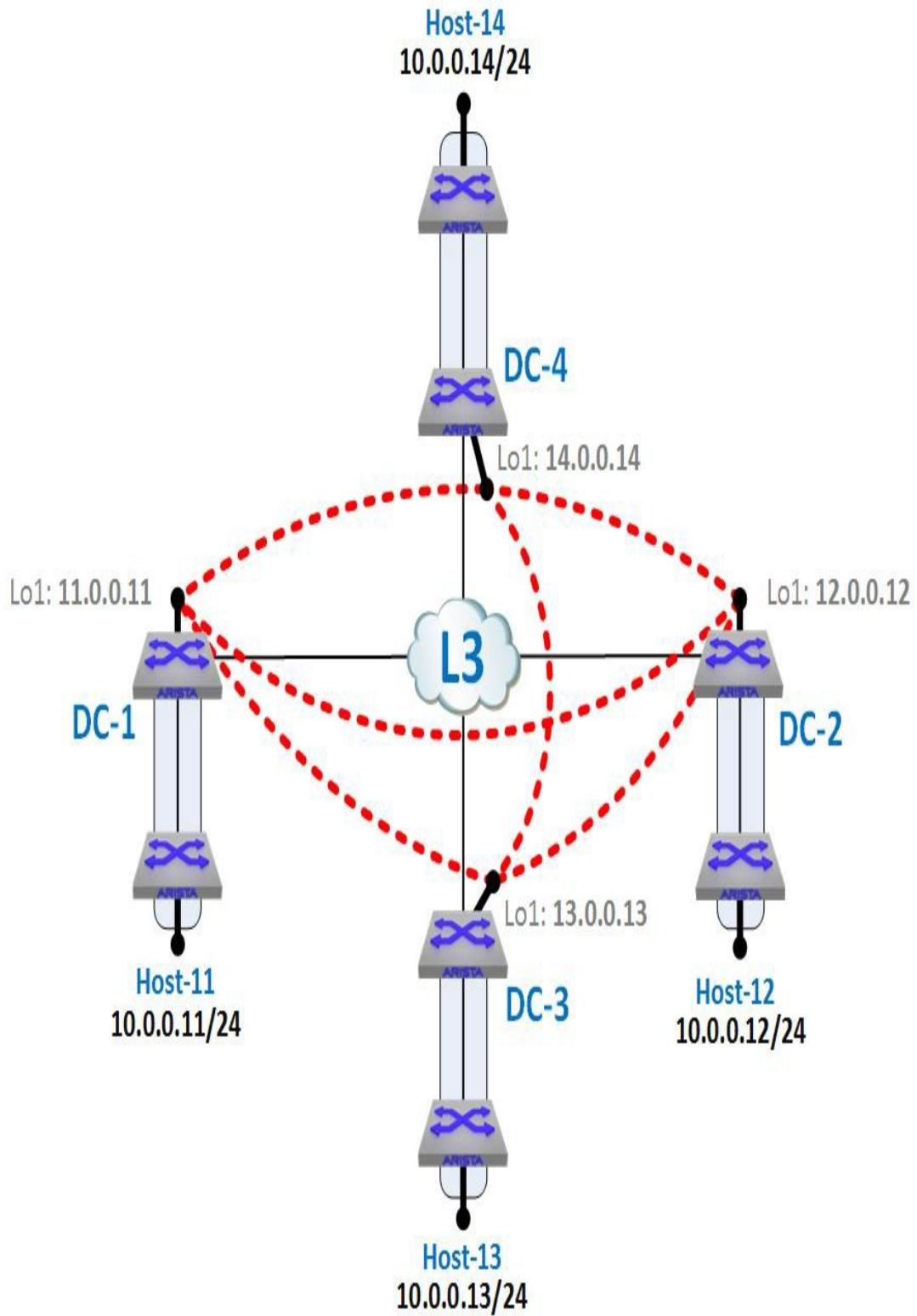


Figure 21-13. VxLAN with DC4 as seen logically

Note that DC4 is a single VTEP and there is really no obvious appearance of MLAG, which is kind of the point of MLAG in the first place. The fact that it can work as a logical VTEP makes it even cooler, though.

I've also added dotted lines that show the VXLAN Segments as they can exist given the VLAN-VNI mappings and distributed flood-lists. Because CVX is distributing the flood-lists and they're all the same, the VTEPs have become fully meshed and any of the hosts on the 10.0.0.x network can ping any of the other hosts in any of the data centers, even though they're separated by L3 networks and even though one of the data centers is using an L2 MLAG design internally.

How does the VXLAN-configured MLAG pair handle a failure, though? Actually, it's really not that exciting and works exactly the way you'd expect. Here's a short list of possibilities:

Upstream link failure

BGP routes over VLAN 5 to the other peer and uses the other link

MLAG peer failure

Normal Link Aggregation Control Protocol rehashing takes care of L2 paths, whereas BGP continues to route accordingly upstream

In our configured network a complex failure could bring DC4 down, and by complex failure I mean the upstream link from DC4-1 fails while DC4-2 peer also fails. That would put DC4 offline, which is why the upstream should be a four-way Equal-Cost MultiPathing link, but now we're getting deeper into the resiliency aspect of network design, which is not really what this chapter is for.

Conclusion

There are myriad other advanced topics relating to VXLAN that I'm not going to cover here. This chapter is one of the largest in the book already (and that's without EVPN!), so topics such as L3 routing with VXLAN, VXLAN pseudo-wire for virtual data centers, data-center interconnect, and a bunch of other buzzword-heavy topics have not been included. Because this book is aimed primarily at the new Arista user and the second edition is already getting too large, such advanced topics are for another venue.

VXLAN is a powerful tool that can solve problems that traditional networking simply cannot. Even though there are certainly other similar solutions out there that claim to do the same thing, VXLAN as an open standard seems to be where the industry has chosen to go.

Chapter 22. Email

Email on a switch? Hell yes! Arista switches allow emails to be sent from the EOS command line, from Bash, from scripts, and from all sorts of interesting places. After you see this in action, you'll wonder how you ever lived without it. Ever have to copy the output of a `show tech` from `flash:`, to a TFTP server, and then to your laptop? You'll never need to go through that nonsense again with email configured on your Arista switch. Ever copy and paste from the screen, only to discover that your scroll-back buffer wasn't big enough? With email on an Arista switch, just email the output directly to your (or anyone's) inbox. But enough hype, let's dig in and see how it's done.

Arista switches contain an email configuration mode that is accessed with the `email` command:

```
Arista#conf
Arista(config)#email
```

When you're there, type a question mark (?) and see what's available:

```
Arista(config-email)#?
auth      Email account authentication
from-user Send email from this user
server    Email relay
tls       Require TLS
-----
comment  Up to 240 characters, comment for this mode
default   Set a command to its defaults
exit     Exit from Email configuration mode
help     Description of the interactive help system
no      Negate a command or set its defaults
```

```
show      Show running system information
!!        Append to comment
```

In its simplest form, mail on an Arista switch requires configuration of a *from* address and an email server to send through. This is done by using the **from-user** and **server** commands. Here, I'll configure the **from-user** to be `gad-lab-arista@gad.net`, and the server to be `192.168.1.200`. If DNS is configured, I could also use a fully qualified domain name such as *mail.gad.net*:

```
Arista(config-email)#from-user gad-lab-arista@gad.net
Arista(config-email)#server 192.168.1.200
```

While within the email configuration mode, the command **show active** displays what's currently configured for email. With the addition of Virtual Routing and Forwarding (VRF) support, the **server** command now shows the default VRF, unless otherwise configured:

```
Arista(config-email)#show active
email
  from-user gad-lab-arista@gad.net
  server vrf default 192.168.1.200
```

For more advanced scenarios, email in EOS supports username and password authentication using the cleverly named **username** and **password** commands:

```
Arista(config-email)#auth username gad
Arista(config-email)#auth password ILikePie
```

If a password is entered in plain text, as I've done here, the switch will convert it to an encrypted string. **Show active** displays this encrypted string, as will the configuration:

```
Arista(config-email)#sho active
email
  from-user gad-lab-arista@gad.net
  server vrf default 192.168.1.200
  auth username gad
  auth password 7 MHTq67ztWA9dQ0fAwOW0qQ==
```

WARNING

Passwords encrypted within configurations using MD5 are not very secure, and MD5 is what's used in the email configuration section even on EOS 4.21.1F where sha512 is used for local EOS user passwords. Remember that given this configuration, the username and password will be sent over the network in clear text, as well.

If your mail server supports Transport Layer Security (TLS), you can enable that with the `tls` command:

```
Arista(config-email)#tls
```

My lab is not set up for TLS, so it won't show up in later command outputs. TLS will solve the problem of passwords being sent in clear text, so it's a recommended solution to use wherever possible.

With my email set up, I'll now flex my new power by sending the output of a command to my inbox. I can do this with any `show` command by using the pipe (vertical bar) character followed by the word *email*. Note that this option does not show up if you search for it:

```
Arista#sho run | ?
LINE      Filter command by common Linux tools such as grep/awk/sed/wc
append   Append redirected output to URL
begin    Begin with the line that matches
exclude  Exclude lines that match
include  Include lines that match
json     Produce JSON output for this command
```

```
no-more  Disable pagination for this command
nz       Include only non-zero counters
redirect Redirect output to URL
section  Include sections that match
tee      Copy output to URL
```

Rest assured, though, that it works. By now it shouldn't surprise you that email is actually a command in Bash that's referenced from EOS. To see the possible options, drop to Bash and issue the `email --help` command:

```
Arista#bash

Arista Networks EOS shell

[admin@Arista ~]$ email --help
Usage: email -- send email through the configured SMTP server

Options:
  -h, --help                  show this help message and exit
  -a ATTACHMENT, --attachment=ATTACHMENT
                                send the named file as an attachment
  -b, --binary                force encoding attachments as binary
  -d, --debug                 debug interaction with SMTP server
  -i, --interactive           force interactive mode even if stdin is not a TTY
  -r REF, --ref=REF            specify case ref
  -s SUBJECT, --subject=SUBJECT
                                specify subject
  --sysname=SYSNAME           specify Sysdb sysname
```

Let's get back to EOS and try some of those. First, I pipe the output of the `show run` command to my email with a subject of *Show Run*. I specify a subject for the email by using the `-s` flag and then list the email address of the intended recipient:

```
Arista(config-email)#sho run | email -s "Show Run" gad@gad.net
Arista(config-email)#

```

No output is displayed because it's all been redirected to the email

program. A quick jump over to my email client, and there's the email! Note that the output is stored as an attachment and is not sent in the body of the email:

```
Date: Fri, 8 Sep 2017 18:40:20
From: gad-lab-arista@gad.net
To: gad@gad.net
Subject: [SPAM] Show Run
Parts/Attachments:
 1 Shown    2 lines  Text
 2          3.8 KB   Application
-----
```

see attachment

```
[ Part 2, Application/OCTET-STREAM 3.8 KB. ]
[ Cannot display this part. Press "V" then "S" to save in a file. ]
```

This time, I send the output of the command `show interface e24` to my email, but without specifying a subject. Without a subject specified, a generic subject is inserted on my behalf:

```
Arista(config-email)#sho int e1 | email gad@gad.net
```

Here is the resulting email, with the subject line in bold, and, yes, check your spam folders if you don't see it in your inbox! This was triggered as spam by my system because the email server does not have reverse DNS configured.

```
Date: Fri, 8 Sep 2017 18:42:07
From: gad-lab-arista@gad.net
To: gad@gad.net
Subject: [SPAM] Support email sent from the switch
Parts/Attachments:
 1 Shown    2 lines  Text
 2          46 KB   Application
-----
```

see attachment

[Part 2, Application/OCTET-STREAM 46 KB.]
[Cannot display this part. Press "V" then "S" to save in a file.]

The email feature used to send the command output in the body of the message (see *Arista Warrior*, first edition), but now the default is to send it as an attachment. Here is the file contained in the last email that was sent:

```
Ethernet1 is up, line protocol is up (connected)
  Hardware is Ethernet, address is 001c.7390.93d0 (bia 001c.7390.93d0)
  Description: [ ESXi ]
  Ethernet MTU 9214 bytes , BW 1000000 kbit
  Full-duplex, 1Gb/s, auto negotiation: on, uni-link: n/a
  Up 24 days, 23 hours, 50 minutes, 53 seconds
  Loopback Mode : None
  3 link status changes since last clear
  Last clearing of "show interface" counters never
  5 minutes input rate 0 bps (0.0% with framing overhead), 0 packets/sec
  5 minutes output rate 636 bps (0.0% with framing overhead), 1
  packets/sec
    0 packets input, 0 bytes
    Received 0 broadcasts, 0 multicast
    0 runts, 0 giants
    0 input errors, 0 CRC, 0 alignment, 0 symbol, 0 input discards
    0 PAUSE input
    1898042 packets output, 194708151 bytes
    Sent 747067 broadcasts, 1150975 multicast
    0 output errors, 0 collisions
    0 late collision, 0 deferred, 0 output discards
    0 PAUSE output
```

Flummoxed by email failures after you've configured your switch for this feature? You can specify the -d option with email, after which you will be rewarded with pages of debug information reflecting every detailed interaction performed by the email process. Let's take a look:

```
Arista#sho int e24 | email -d gad@gad.net
connect: ('192.168.1.200', 25)
```

```
connect: (25, '192.168.1.200')
reply: '220 mail.example.com ESMTP Postfix (Ubuntu)\r\n'
reply: retcode (220); Msg: mail.example.com ESMTP Postfix (Ubuntu)
connect: mail.example.com ESMTP Postfix (Ubuntu)
send: 'ehlo [127.0.0.1]\r\n'
reply: '250-mail.example.com\r\n'
reply: '250-PIPELINING\r\n'
reply: '250-SIZE 30000000\r\n'
reply: '250-VRFY\r\n'
reply: '250-ETRN\r\n'
reply: '250-STARTTLS\r\n'
reply: '250-ENHANCEDSTATUSCODES\r\n'
reply: '250-8BITMIME\r\n'
reply: '250 DSN\r\n'
reply: retcode (250); Msg: mail.example.com
PIPELINING
SIZE 30000000
VRFY
ETRN
STARTTLS
ENHANCEDSTATUSCODES
8BITMIME
[---output truncated--]
```

In this case, everything went through fine, but be warned that this can create a lot of output depending on what's going on. This output would be invaluable during a failure. Here, I've misconfigured the server's IP address in my email configuration in order to generate a failed connection:

```
Arista#sho int e24 | email -d -s "Show Int e24" gad@gad.net
connect: ('1.1.1.1', 25)
connect: ('1.1.1.1', 25)
% Failed to send email: [Errno 101] Network is unreachable
```

Because `email` is actually a Bash command, you can use it for redirecting output in Bash, too. Here, I've redirected the output of `ls -al` to my email address:

```
[admin@Arista ~]$ ls -al | email -s "ls -al" gad@gad.net
```

Conclusion

If you're like me, you'll find yourself using this feature a lot more than you ever thought you would. But then, I've been told there aren't a lot of people quite like me. The worst thing is that after you get used to all these cool Arista features, it can be absolutely maddening to use any other vendor's switch.

Chapter 23. LANZ

If you've read the chapter on buffers (Chapter 3), you know that they can be a benefit or bane, depending on a lot of factors. When buffers in a switch become a problem, it can be very difficult to isolate the problem because there usually aren't detailed counters that show the buffer contents. When running quality of service (QoS) on routers, there are all kinds of commands to run that will show you the status of your buffers, but those buffers are software constructs that take memory from the system. The buffers I'm referring to here are hardware switch interface buffers. Let's dig in, and I'll show you what I mean.

Here's the output from the `show interface` command on an Arista 7280R. As you can see, there is no mention of buffers:

```
Arista-1#sho int e48
Ethernet48 is up, line protocol is up (connected)
  Hardware is Ethernet, address is 2899.3abe.a026
  Description: desc [ Arista-2 ]
  Internet address is 88.1.0.1/30
  Broadcast address is 255.255.255.255
  IP MTU 1500 bytes , BW 10000000 kbit
  Full-duplex, 10Gb/s, auto negotiation: off, uni-link: disabled
  Up 19 hours, 56 minutes, 8 seconds
  Loopback Mode : None
  11 link status changes since last clear
  Last clearing of "show interface" counters 13 days, 20:40:55 ago
  5 minutes input rate 55 bps (0.0% with framing overhead), 0 packets/sec
  5 minutes output rate 60 bps (0.0% with framing overhead), 0
  packets/sec
    74213421 packets input, 110899750528 bytes
    Received 2 broadcasts, 8437 multicast
```

```
0 runts, 0 giants
0 input errors, 0 CRC, 0 alignment, 0 symbol, 0 input discards
0 PAUSE input
76733308 packets output, 114571891285 bytes
Sent 7254 broadcasts, 97979 multicast
0 output errors, 0 collisions
0 late collision, 0 deferred, 4372620 output discards
0 PAUSE output
```

Why is there no mention of buffers? I didn't write the code, but I can guess that the status of the interface buffers changes at the microsecond level, so between the time you begin moving to press the Enter key and the time you finish, the status of the buffers likely changed, so any information put into the output of the `show interface` command would be woefully outdated by the time it's presented.

Microbursts Visualized

Let's look at what a microburst looks like from a traffic usage pattern. Traditionally network interfaces are monitored using Simple Network Management Protocol (SNMP) tools such as Solarwinds, MRTG, HP OpenView, or the like. These tools usually poll once every five minutes, though some environments poll more often. Polling more often gives more refined graphs due to having more granular data at the cost of increased SNMP traffic on the network. Figure 23-1 depicts a typical interface utilization graph.

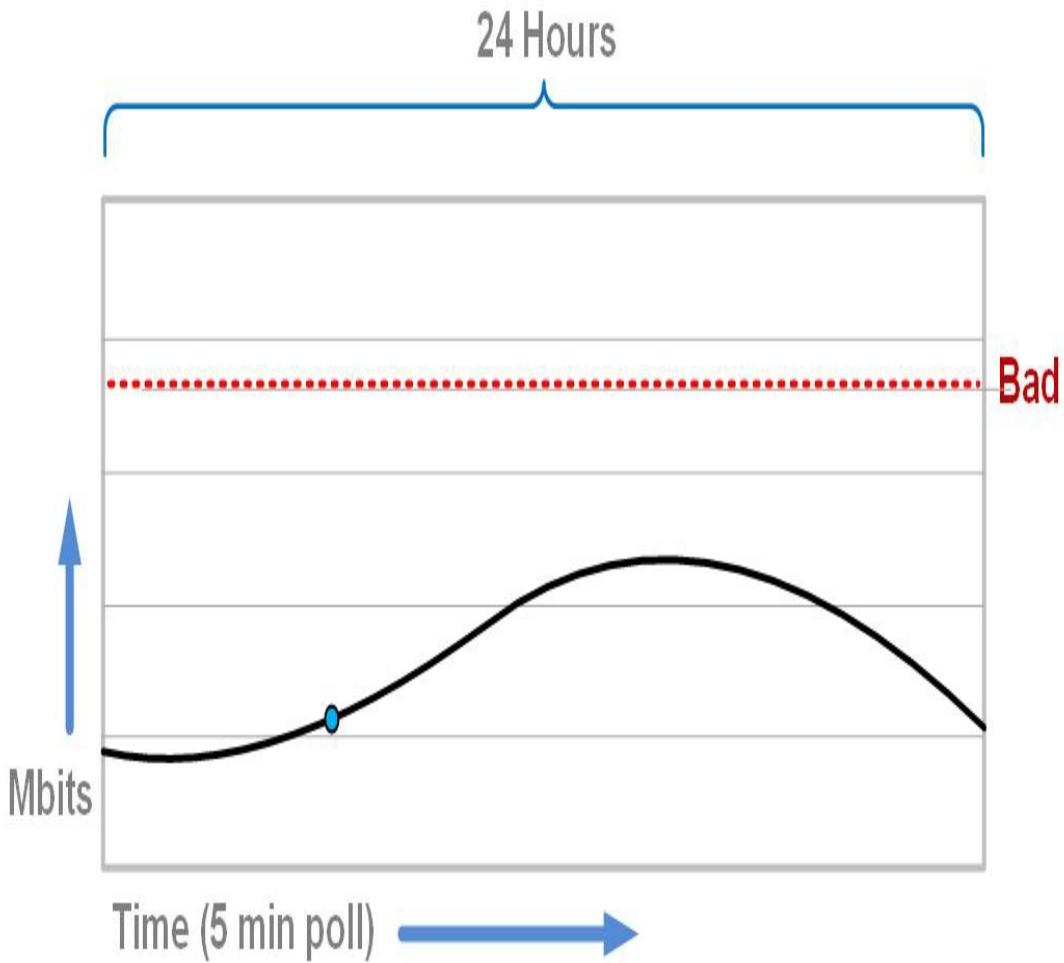


Figure 23-1. Typical interface utilization graph with five-minute SNMP polling

If someone were to come to me and say that the network sucks, I would probably look at the graph and say, “Nope!” The graphs are all below the line that says *bad*, so go away and bother someone else. The problem is that the graph is not as accurate as we’d like to believe.

The tools producing these graphs do so by polling the devices using something called an *SNMP GET*. A typical request might be “get the number of bytes that have output from this interface.” This GET will return an integer that is literally the number of bytes that have been sent out the interface since the switch was last booted. And yes, this

can be a ridiculously large number. You can see these numbers using the `show interface [int-name] counters` command. Here's an example:

```
Arista-2#sho int e48 counters outgoing
Port      OutOctets   OutUcastPkts   OutMcastPkts   OutBcastPkts
Et48      139617360586      92095180          5451          0
```

Assuming the default interval, the graphs are created by polling (doing an SNMP get) every five minutes. The latest value is subtracted from the value from the last poll, and that number is put into the graph. It's literally that simple (our graph is showing Mbits, so some additional math is performed). Let's see how that would shake out using the counters after five minutes of traffic on this interface:

```
Arista-2#sho int e48 counters outgoing
Port      OutOctets   OutUcastPkts   OutMcastPkts   OutBcastPkts
Et48      144502369406      95317527          5461          0
```

Having grabbed the data twice, if we subtract the two `OutOctet` numbers, we get the number of octets (bytes) sent out by that interface over that interval of time. Let's use Bash's `expr` command to do the math:

```
Arista-2#bash expr 144502369406 - 139617360586
4885008820
```

Adding some commas (sorry, Europeans) for formatting, we get the more human readable 4,885,008,820 so 4.8 GigaBYTES (remember, we measured how many octets were output from this interface over five minutes). If we do some more math, convert that to bits

```
Arista-2#bash expr 4885008820 * 8
```

```
39080070560
```

and then divide that by the number of seconds in five minutes (300), we get the following:

```
Arista-2#bash expr 39080070560 / 300
130266901
```

Adding commas, we get 130,266,901, which is 130.26 Mbps. For the five-minute period between the two SNMP gets, the switch output 130.25 Mbps.

But did it, really?

The problem with this method is granularity. We've basically reduced the entire five-minute period to two values and subtracted them to put a number into a graph, but the network did not send a steady state of 130.25 Mbps traffic during those five minutes, and depending on the nature of the traffic on the network, those SNMP graphs that we've all been trusting for decades might be misleading us.

In [Figure 23-1](#), you'll notice a small circle on the graph. At that point in time, I was able to monitor the outbound interface every 1/1,000 of a second. With millisecond accuracy, I collected what the traffic looked like and put it into the graph shown in [Figure 23-2](#).

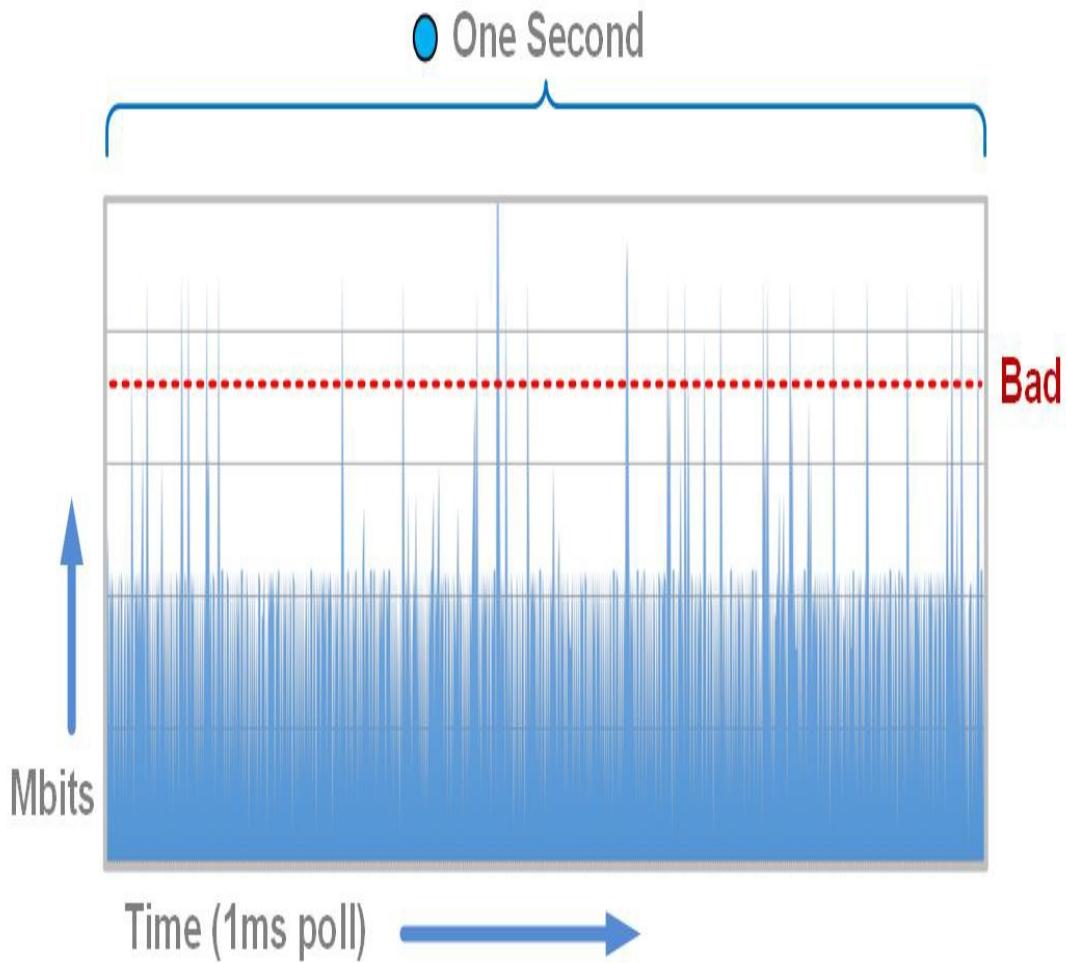


Figure 23-2. Spot from previous graph zoomed in to impossible one-second interval

Holy moly! See those spikes that are crossing the *bad* line? Those are microbursts, and they're causing people to think that the network sucks even though the SNMP graphs all look good. Believe it or not, this example doesn't even scratch the surface of how bad it can be! There can be flat-out buffer-full oversubscription on the interface for short durations that cause packets to be dropped while not even showing up as a blip on the SNMP graphs.

After living through this a couple of times, you learn to sort of smell when it's happening, but having lived through microburst events that

caused performance problems, I can tell you that I would have given my right arm for useful tools that would give me graphs like the 1-second example shown earlier. OK, maybe not my entire right arm, but believe me when I say that the lack of visibility into these buffers is extremely frustrating. I've been through enough of these problems that I swear I can smell them, but being unable to prove them makes it difficult to coax management to spend money to fix them. If only someone would make a switch that had real troubleshooting tools at the interface buffer level! Enter LANZ.

Latency analyzer (LANZ) is Arista's solution to this problem. On certain Arista switches, the Application-Specific Integrated Circuits (ASICs) allow visibility into the interface buffers at a heretofore unheard-of reporting granularity of less than one millisecond. For those of you who can't keep the whole micro/nano/pico thing straight, a millisecond is one-thousandth of a second.

NOTE

I said this is a result of an ASIC feature. This means that this is a hardware-based feature and, as such, is available only on the Arista switches that use the ASICs that support it. See the Arista feature matrix or talk to your sales team for details. Also note that because this is a hardware feature, LANZ is not available in any version of vEOS.

That's right, we can now get reports that show the status of our interface buffers 1,000 times per second. Cool, huh? Sounds like too much information to me, but I'm a cranky old nerd. Let's take a look and see how it works.

Queue Thresholds

To understand what LANZ is doing I'd like to take a brief detour into simple buffer theory. If you're not familiar with this sort of thing, you've probably seen it before when you configure a serial port (such as a console) using the settings for flow control. One of those settings is called XON/XOFF, which I use for this example, but the principles are fundamentally the same regardless of whether it's RTS/CTS, TCP Source Quench, or Ethernet flow control, though the actual mechanism or configuration might be different. Let's consider a serial port buffer such as the one shown in Figure 23-3.

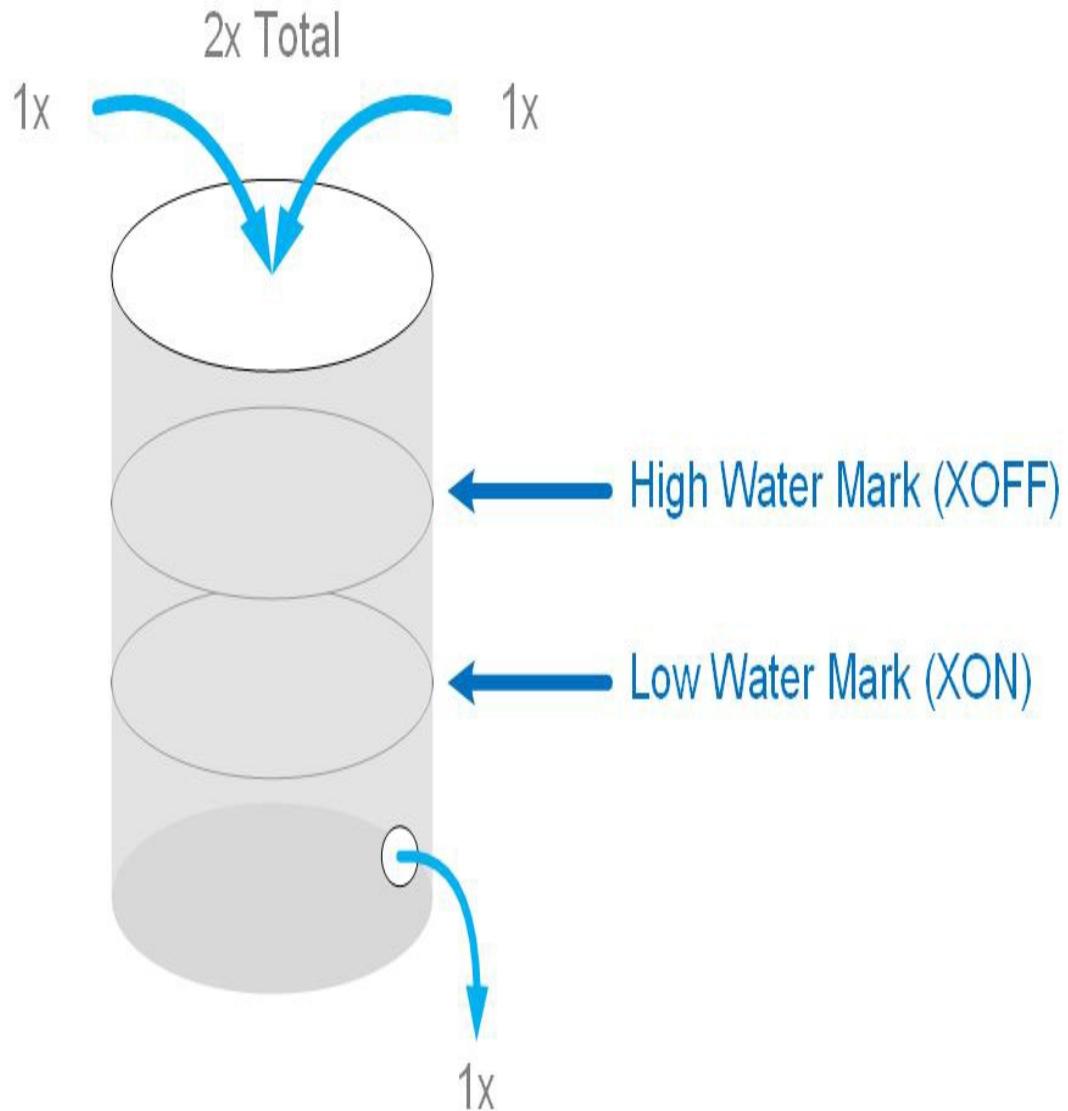


Figure 23-3. Simple serial port buffer

The buffer is shown as a simplistic container, like a 55-gallon drum or a smaller flask. Essentially, 1x worth of water can exit the bottom, but we can easily fill the container with 2x (or more) water. If we continue doing so, the buffer will overflow and water will get all over the floor, and who wants to clean that up? Not me.

To prevent the buffer from overflowing, sensors are installed, and when the water in the container reaches the High-Water Mark, a signal

is sent to the people filling the container, telling them to knock it off. This is the XOFF signal in serial flow control. After the inflow of water stops, the container continues to drain at 1x through the bottom. When the water level lowers to the Low-Water Mark, another signal is sent (this time an XON) to people with the hoses who then begin filling the container again. By letting these sensors inform the people filling the container, the container should never overflow. Again, this is the idea behind most flow-control mechanisms out there.

LANZ is not a flow-control mechanism. LANZ is a reporting tool that allows us to see what's happening within a buffer, but it's important to understand that LANZ does not affect the flow of data through that buffer. The way that it reports on buffer status is conceptually similar, though, so consider Figure 23-4.

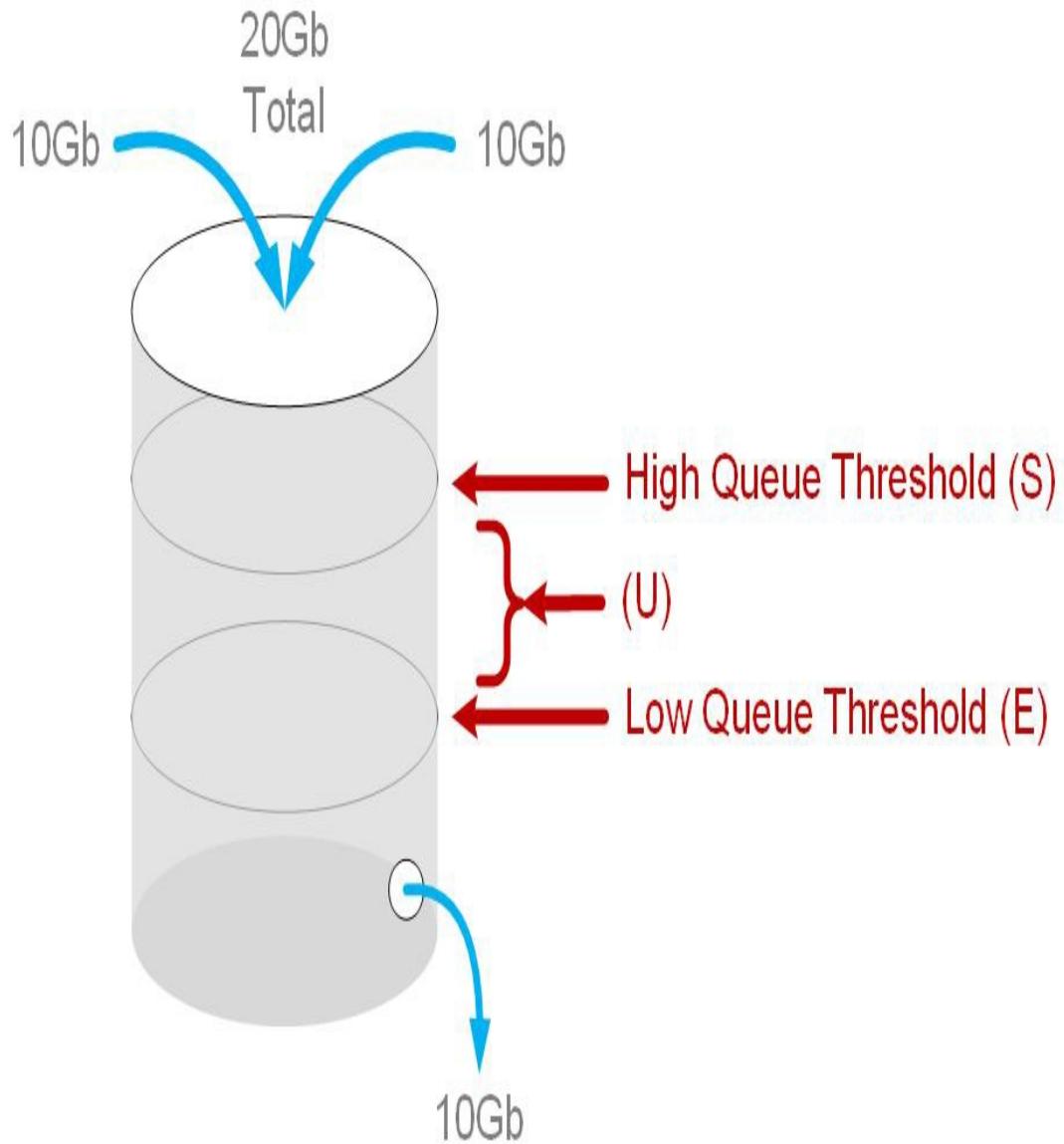


Figure 23-4. LANZ with high- and low-queue thresholds

Here, the container is the buffer for a 10 Gbps Ethernet interface. That buffer can be filled with multiple source interfaces, and if we allow two 10 Gbps Ethernet interfaces to fill the buffer, it will overflow. When the buffer overflows, packets are dropped, and even though there are no mops involved, I'm still the guy who has to clean it up.

With LANZ we configure a High-Queue Threshold, which is similar in

principle to the High-Water Mark except that it does not affect flow. It is the threshold at which, when the buffer fills to that point, LANZ will send a Start (S) record. Hopefully, either through some flow-control mechanism or through organic change, the buffer will begin to deplete, during which time LANZ will send Update (U) records. When the Low-Queue Threshold is met, LANZ will send an End (E) record. Some switches, like the 7280R, will send Polling (P) records as well when in polling mode.

Polling mode is the default state on some Arista switches and differs from *notification mode* in that notification mode uses more hardware resources (which is why it's not the default on those boxes). Internally there a bunch of different ways that LANZ uses the hardware, which is far beyond the scope of this book. Just know that if you're seeing only P records that your switch is in polling mode.

For my first example, I have a very simple network setup, as presented in Figure 23-5. This network comprises an Arista 7280R switch hooked up to another Arista 7280R switch. In the first edition of *Arista Warrior*, I used a Cisco 3750 as the other switch, which caused all sorts of mayhem due to that switch using processed switching for packets destined to the switch. Although amusing, I decided that it was time to embrace the future. Also, back then I had only one old Arista switch to play with, and now I have access to rooms full of them.

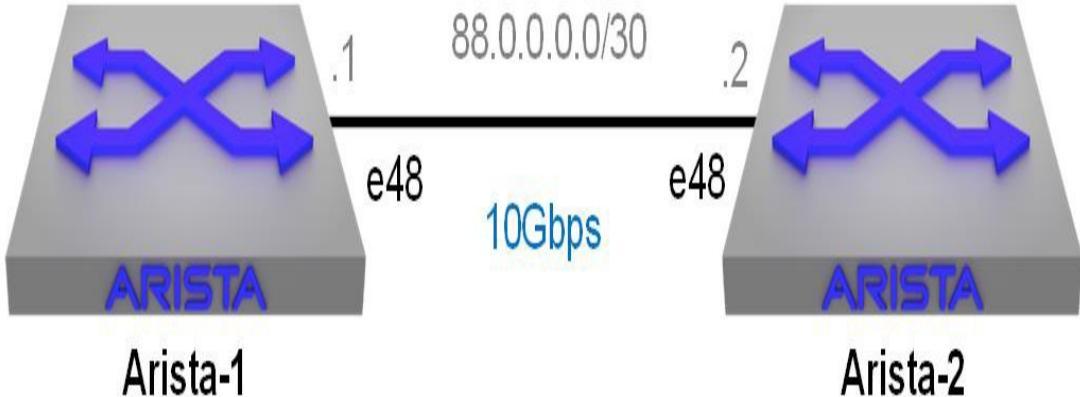


Figure 23-5. A simple LANZ test lab

I've connected these two switches using 10 Gbps Twinax cables. Here are the configurations for the two switches to accomplish this simple design. First, Arista-1:

```
Arista-1(config-if-Et48)#sho run int e48
interface Ethernet48
  description desc [ Arista-2 ]
  no switchport
  ip address 88.0.0.1/30
```

And now Arista-2:

```
Arista-2#sho run int e48
interface Ethernet48
  description [ Arista-1 ]
  no switchport
  ip address 88.0.0.2/30
```

With Arista switches being so damn powerful, there's no way I could overwhelm that 10 Gbps link with simple pings. There are some additional limitations with the way that LANZ works on the modern 7280Rs with their Jericho ASICs versus the way the older switches worked, so I needed a more interesting solution. I had toyed around with disabling Spanning Tree and making loops with multiple

interfaces, but that was too complicated for various reasons. Arista trainer Rich Parkin thought up a cool scenario that allows us to deliver a 10 Gbps amount of traffic easily while lowering the effective speed of one of the interfaces in order to trigger buffering that we can see with LANZ. Figure 23-6 shows that design.

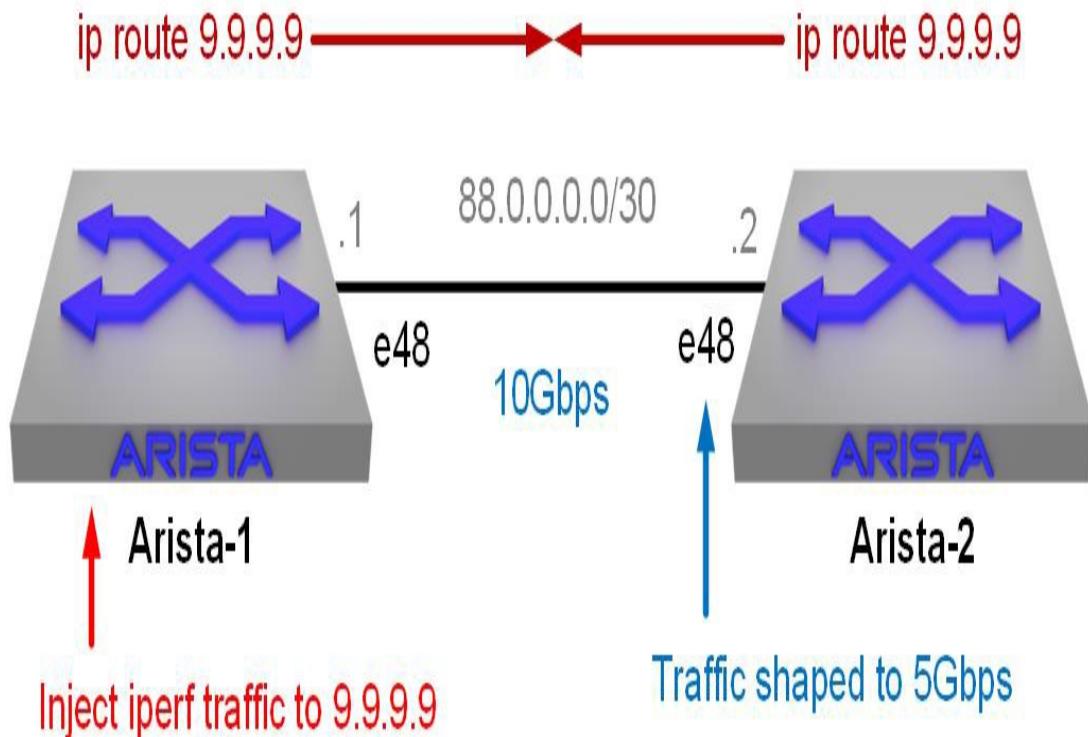


Figure 23-6. A simple LANZ test lab with routing loop

Simply put, we're going to make a routing loop between the two switches to a destination IP address 9.9.9.9. We're then going to use traffic-shaping to make the e48 interface on Arista-2 behave like a 5 Gbps interface. We'll use `iperf` (more on that in a minute) to inject traffic destined for 9.9.9.9 and then sit back and watch while the mayhem ensues.

Here is the configuration needed to set up the lab with these new

requirements:

Arista-1 is simple:

```
Arista-1#sho run | grep rout
ip route 9.9.9.9/32 88.0.0.2
ip routing
```

Arista-2 has a little bit more to it, but first, here are the matching routing commands:

```
Arista-2(config)#sho run | grep rout
ip route 9.9.9.9/32 88.0.0.1
ip routing
```

That accomplishes the routing loop, but we need to configure e48 with some additional values in order to throttle the traffic:

```
Arista-2(config)#sho run int e48
interface Ethernet48
  description [ Arista-1 ]
  load-interval 1
  no switchport
  ip address 88.0.0.2/30
  shape rate 5000000
```

The **load-interval** command has no effect on traffic flow and is just there for ease of reporting. The real magic here is the **shape rate 5000000** command, which effectively makes the interface behave as if it were only a 5 Gbps port. Although the internals can be a tad more complicated than that, the result is that with 10 Gbps of traffic trying to be sent back to Arista-1, the traffic will need to be buffered, which is what we want to see with LANZ.

With the basic network configured, let's go in and configure LANZ.

We're only going to do on Arista-2 because that's where the buffering is set to happen. First, we need to enable the feature globally. We do this by using the `queue-monitor length` command:

```
Arista-2#queue-monitor length
```

At this point, LANZ is running and will record information when the proper thresholds are met. Next, we configure LANZ on the interface connected to the other switch. This is done with a similar command that needs to include two threshold values, the lower threshold and the upper threshold:

```
Arista-2(config)#int e48
Arista-2(config-if-Et48)#queue-monitor length thresholds 40962
10241
```

Where did I get those numbers from? Well, they're the High-Queue threshold and the Low-Queue threshold for the interfaces and are the lowest allowable values on this platform:

```
Arista-2(config-if-Et48)#queue-monitor length thresholds ?
<40962-524288000> high threshold in units of bytes

Arista-2(config-if-Et48)#queue-monitor length thresholds 40962 ?
<10241-524288000> low threshold in units of bytes

Arista-2(config-if-Et48)#queue-monitor length thresholds 40962
10241
```

On the older boxes with different ASICs, like the 7150s, the thresholds were set using buffer segments instead of bytes, and you could configure much lower numbers like this:

```
Arista-7150(config-if-Et5)#queue-monitor length thresholds 2 1
```

The 7280s also allow just a High-Queue threshold to be set, so be careful if you're used to using autocompletion because the word threshold (singular) is what you'll end up with and you won't be able to put in the Low-Queue threshold.

With LANZ configured you can check the status with the `show queue-monitor length status` command:

```
Arista-2(config-if-Et48)#sho queue-monitor length status
queue-monitor length enabled
queue-monitor length packet sampling is disabled
queue-monitor length update interval in micro seconds: 5000000
Per-Interface Queue Length Monitoring
-----
Queue length monitoring is enabled
Queue length monitoring mode is:
    FixedSystem    polling
Maximum queue length in bytes : 524288000
Port thresholds in bytes:
Port      High threshold  Low threshold      Warnings
Cpu          65536          32768
Et1          5242880         2621440
Et2          5242880         2621440
Et3          5242880         2621440
Et4          5242880         2621440
[--output removed--]
Et46         5242880         2621440
Et47         5242880         2621440
Et48          40962          10241
Et49/1       5242880         2621440
Et49/2       5242880         2621440
Et49/3       5242880         2621440
[--output removed--]
Et54/1       5242880         2621440
Et54/2       5242880         2621440
Et54/3       5242880         2621440
Et54/4       5242880         2621440
```

With this command, we can see a bunch of information about LANZ including the queue thresholds. We can see that we've altered the

values on e48 from the defaults. We can also see that this switch is configured for polling mode, but we're going to leave it there for now. Enough configuration; let's abuse some buffers!

In the first edition of *Arista Warrior*, I had to resort to sending ping packets from Bash to the remote switch like this:

```
Arista-1#bash
Arista Networks EOS shell

[admin@Arista-1 ~]$ ping -s 15000 -c 1000 88.0.0.2 > /dev/null &
[1] 9054
[admin@Arista-1 ~]$ ping -s 15000 -c 1000 88.0.0.2 > /dev/null &
[2] 9057
[admin@Arista-1 ~]$ ping -s 15000 -c 1000 88.0.0.2 > /dev/null &
[3] 9060
```

That certainly works, but it's lame because with today's EOS, the `iperf` tool is installed by default! With the interfaces configured, LANZ turned on, and our routing loop ready to go, let's get crazy and send a 1 Gb barrage of data into the loop. We'll do this using `iperf` from Bash on Arista-1. This command will attempt to send 1 Gbps of traffic using User Datagram Protocol (UDP) to 9.9.9.9:

```
[admin@Arista-1 ~]$ iperf -uc 9.9.9.9 -b 1G
-----
Client connecting to 9.9.9.9, UDP port 5001
Sending 1470 byte datagrams
UDP buffer size: 208 KByte (default)
-----
[ 3] local 88.0.0.1 port 54631 connected with 9.9.9.9 port 5001
[ ID] Interval Transfer Bandwidth
[ 3] 0.0-10.0 sec 968 MBytes 812 Mbits/sec
[ 3] Sent 690431 datagrams
[ 3] WARNING: did not receive ack of last datagram after 10 tries.
```

Within a couple of seconds, the interface on Arista-2 is sending at its maximum shape-rate:

```
Arista-2#sho int e48 | grep rate
 1 second input rate 5.69 Gbps (57.7% with framing overhead), 469536
pkts/sec
 1 second output rate 4.87 Gbps (49.3% with framing overhead), 401408
pkts/sec
```

NOTE

This is the power of Linux that you've heard so much about. It's absolutely worth learning because you can do things like this, which is super helpful especially when you're writing a book (or any other documentation) and the output is so wide and complicated that the reader (and my editor) would complain that it wraps and then looks impossible to understand.

Because the routing loop is self-limiting due to Time to Live (TTL), after a few seconds the interface drops back down to zero:

```
Arista-2#sho int e48 | grep rate
 1 second input rate 0 bps (0.0% with framing overhead), 0 packets/sec
 1 second output rate 0 bps (0.0% with framing overhead), 0 packets/sec
```

Let's see what LANZ saw during this assault:

```
Arista-2#sho queue-monitor length e48

Report generated at 2019-01-16 23:00:59
S-Start, U-Update, E-End, P-Polling, TC-Traffic Class
* Max queue length during period of congestion
Type Time           Intf(TC) Queue      Duration  Ingress
                  Length          Port-set
                  (bytes)   (usecs)
-----
-----
P    0:02:38.31162 ago Et48(1)  38936576  12012760  Et1-8,17-20,29-32,41-
48,51/1,
```

Note that some of the rightmost output has been truncated because it's just too much to fit and I didn't want it to wrap. The reason for that odd output is sort of interesting, though, so let's look at that last column, which is the `Ingress Port-set`. That's a very strange interface range that I absolutely did not configure. After talking to the developers at Arista, I finally understood what this represented, which is all of the interfaces that are on the same ASIC core as the interface in question. Um...what?

On the Jericho-based switches like the 7280R I'm using here, the ASIC comprises multiple cores just like the cores you'd find in your PC's CPU. The front-panel interfaces are distributed across multiple cores in the ASIC, and this is reporting the other interfaces found on that core. Another way that you can see this is by using the `show platform` command. I'm going to do some Linux wizardry here to show you a vastly simplified output. If you'd like to see the entire output, use the `show platform jericho mapping` command on your 7280R or other Jericho-based Arista switch:

```
[admin@Arista-2 ~]$ Cli -p 15 -c "show platform jericho mapping"
|
{ head -n3 ; grep Ether; } | awk '{printf "%-12s %s\n", $1,
$4}'
```

Port	Core
Ethernet1	0
Ethernet2	0
Ethernet3	0
Ethernet4	0
Ethernet5	0
Ethernet6	0
Ethernet7	0
Ethernet8	0
Ethernet9	1

```
Ethernet10    1
Ethernet11    1
Ethernet12    1
Ethernet13    1
Ethernet14    1
Ethernet15    1
Ethernet16    1
Ethernet17    0
Ethernet18    0
Ethernet19    0
Ethernet20    0
[-- output truncated --]
```

If that huge statement hurts your head, here's what it's doing:

1. Issue the CLI command `show platform jerich mapping`
2. Include the first three lines of the header
3. Include only lines after the header that include the string *Ether*
4. Rearrange the output so that only the first and fourth fields are printed with the first field left-justified in 12 spaces

That output shows which of the front-panel interfaces are associated with which Jericho ASIC core, which just happens to line up with what we saw in LANZ (more AWK!):

```
Arista-2#sho queue-monitor length e48 | grep Et48 | awk '{print
$7}'
Et1-8,17-20,29-32,41-48,51/1,52/1,53/1
```

Back to our LANZ output, the leftmost column has a P in it, which means that the switch is in *Polling* mode. There are two modes on some Arista switches including this 7280R. They are *polling* and *notification*, with *polling* being the default. Polling mode reports whether there was congestion over the past second with P records, whereas notification mode is much more granular and sends the

aforementioned Start, Update, and End records.

Let's change the switch to use notification mode:

```
Arista-2(config)#queue-monitor length notifying
```

After triggering another barrage of packets, here's the renewed traffic data from `show interface`:

```
Arista-2#sho int e48 | grep rate
 1 second input rate 5.16 Gbps (52.3% with framing overhead), 425683
 pkts/sec
 1 second output rate 4.41 Gbps (44.7% with framing overhead), 364003
 pkts/sec
```

Let's see how the output differs when using notification mode:

```
Arista-2(config)#sho queue-monitor length e48
```

```
Report generated at 2019-01-18 21:04:41
S-Start, U-Update, E-End, P-Polling, TC-Traffic Class
* Max queue length during period of congestion
Type Time           Intf(TC) Queue      Duration Ingress
                           Length          Port-set
                           (bytes)      (usecs)
-----  -----
E  0:00:54.17983 ago  Et48(1)  38936576* 12024591  Et1-8,17-20,29-32,41-
48,51/1,
U  0:00:54.17983 ago  Et48(1)  31966544      N/A    Et1-8,17-20,29-32,41-
48,51/1,
U  0:00:56.18240 ago  Et48(1)  38936576      N/A    Et1-8,17-20,29-32,41-
48,51/1,
U  0:00:57.18424 ago  Et48(1)  38936576      N/A    Et1-8,17-20,29-32,41-
48,51/1,
U  0:00:59.18937 ago  Et48(1)  38936576      N/A    Et1-8,17-20,29-32,41-
48,51/1,
U  0:01:01.19334 ago  Et48(1)  38936576      N/A    Et1-8,17-20,29-32,41-
48,51/1,
U  0:01:02.19565 ago  Et48(1)  38936576      N/A    Et1-8,17-20,29-32,41-
48,51/1,
U  0:01:04.19982 ago  Et48(1)  38936576      N/A    Et1-8,17-20,29-32,41-
```

```
48,51/1,  
S 0:01:06.20442 ago Et48(1) 38936576 N/A Et1-8,17-20,29-32,41-  
48,51/1,  
P 0:02:38.31162 ago Et48(1) 38936576 12012760 Et1-8,17-20,29-32,41-  
48,51/1,
```

As you can see there's a lot more information. Looking from the bottom up, we can see the P record from our last test and then a Start record above it. Next, going up the list we see seven Update records, which show the status of the buffer every two seconds. Lastly, on the top, we see an End record, which indicates the end of the congestion event along with the included duration of the event in microseconds.

All of those numbers are nice, but I'd sure like to see all that data in a graph. Luckily there are a few ways to accomplish this. The first is through the use of CloudVision Portal (CVP), which shows a dizzying array of useful graphs including LANZ information. The data can also be streamed using the Google Protocol Buffer format, but the way I've used LANZ in the field when CloudVision was unavailable is to convert the output to comma-separated values (CSV) format by using the `csv` keyword on the `show queue-monitor length` command. Note that with the `csv` keyword, the oldest samples are displayed first, which is the opposite of what we've seen previously:

```
Arista-2(config)#sho queue-monitor length csv  
  
Report generated at 2019-01-18 21:18:42  
P,2019-01-16 22:44:29.37126,Et48(1),38936576,12021335,1,"Et1-8,17-20,29-  
32,41..."  
P,2019-01-16 22:45:56.47697,Et48(1),38936576,11013245,1,"Et1-8,17-20,29-  
32,41..."  
P,2019-01-16 22:46:31.51801,Et48(1),38936576,12014296,1,"Et1-8,17-20,29-  
32,41..."  
P,2019-01-16 22:50:09.76701,Et48(1),38936576,41049021,1,"Et1-8,17-20,29-  
32,41..."  
P,2019-01-16 22:51:09.83819,Et48(1),38936576,11012865,1,"Et1-8,17-20,29-
```

```
32,41..."  
P,2019-01-16 22:51:33.86530,Et48(1),38936576,12015105,1,"Et1-8,17-20,29-  
32,41..."  
P,2019-01-16 22:51:59.89610,Et48(1),38936576,12015291,1,"Et1-8,17-20,29-  
32,41..."  
P,2019-01-16 22:52:20.92150,Et48(1),38936576,12013686,1,"Et1-8,17-20,29-  
32,41..."  
P,2019-01-16 22:52:46.95214,Et48(1),38936576,12014028,1,"Et1-8,17-20,29-  
32,41..."  
P,2019-01-16 22:53:08.97880,Et48(1),38936576,11013511,1,"Et1-8,17-20,29-  
32,41..."  
P,2019-01-16 22:53:26.99921,Et48(1),38936576,12015860,1,"Et1-8,17-20,29-  
32,41..."  
[--- output truncated ---]
```

I suggest that if you use this command, you either pipe it to `more` or redirect it to file. I suggest this because using the `csv` keyword will report the last 100,000 *over-threshold events*, which is a lot of data to watch on the console. To redirect the output to a file, use the `>` operator just like you would in Linux:

```
Arista-1#sho queue-monitor length csv > flash:CSV-GAD.txt
```

NOTE

If you want this file to survive a reboot, you'll need to place it somewhere on `flash:`, `drive:`, or a USB flash drive.

Importing LANZ CSV output into Microsoft Excel is how I created the one-second utilization graph earlier in this chapter, and while creating Excel charts from data gathered at the CLI is fun, it is limited in its usefulness. If your network is massively congested, the last 100,000 samples might be a small amount of time. Remember that this file can contain the buffer information for every interface on the switch, and, as

we've seen, in a congested network, these entries can add up quickly depending on how you have your thresholds set. What would be more useful would be the ability to stream all this data elsewhere. Luckily, the folks at Arista had this in mind when they built this feature.

Conclusion

Though LANZ is limited to certain switches due to the ASICs used in them, as ASICs mature, more and more of the Arista product portfolio supports the feature. Is it useful in the real world? Absolutely! I've used it to prove microbursting on backup networks and storage networks, and, let me tell you, when you can perform troubleshooting like this, you quickly become the hero. I've also found that doing this kind of troubleshooting tends to make the people who sign the checks realize why Arista switches are the smart choice in a modern network.

Chapter 24. Scheduler

I was working in a data center, building out an Arista network, when I decided that I wanted to automate a command that should run every five minutes. Knowing that these super-cool Arista switches run Linux, I dropped into Bash and proceeded to muck around with *cron*. I couldn't get it to work, became frustrated, and called my Arista sales engineer, who asked, "Why not just use *schedule*?"

Schedule is a feature that's been around since version 4.5 of EOS that allows the regular scheduling of commands. The cool part of this feature is that it's completely configured from the command-line interface (CLI), so you don't need to spend any time swearing at *cron*. In this chapter, we take a look at this powerful tool.

The **schedule** command is the root of just about everything we're going to do in this chapter. It's simple to use, and the question mark (?) along with tab completion will get you most of what you want to know about its function:

```
Arista(config)#schedule ?
WORD      Scheduled job name
config    Set CLI scheduler configuration parameters
```

The only configuration options as of EOS 4.21.1F are **max-concurrent-jobs**, which you can set from 1 to 4, with the default being 1, and **prepend-hostname-logfile**, which does exactly what it sounds like it does and is now the default behavior:

```
Arista(config)#schedule config max-concurrent-jobs ?  
<1-4> Maximum number of concurrent jobs
```

To create a scheduled job, you first must specify a name for the job. Because I'm the writer and I love my daughter, I'll use the name Colleen for my job. There are some cool new options with schedule that didn't exist back in the dark ages when I wrote the first edition of *Arista Warrior*:

```
Arista(config)#schedule Colleen ?  
at           Set the start time of the schedule  
interval    Set interval for CLI command execution  
now         Set the start time of the schedule to now
```

The interval is the amount of time in minutes to wait between each iteration of the job. If you want the job to run every five minutes, the interval would be 5. In version 4.21.1F, acceptable values range from 2 to 1440:

```
Arista(config)#schedule Colleen interval ?  
<2-1440> Interval in minutes for CLI command execution
```

Note that this is a significant change from earlier versions in which the range was 1 to 1440. Although that might not seem all that significant, the change is a result of the interval needing to now be at least twice the value of the timeout, which defaults to 30 seconds:

```
Arista(config)#schedule Colleen interval 5 timeout ?  
<1-480> Timeout in minutes for CLI command execution
```

In fact, if you set the interval to a value that is not greater than the timeout, the schedule will not be accepted:

```
Arista(config)#schedule Colleen interval 5 max-log-files 5
```

```
command sho ver
! Schedule a command starting in past
% For job colleen, Interval 5 should be greater than the timeout 30
```

Also, that message, `Schedule a command starting in past`, is a cosmetic bug that happens when you don't specify an `at`, which I cover in a minute. Let's set a timeout of 4:

```
Arista(config)#schedule Colleen interval 5 timeout 4 ?
max-log-files  Set maximum number of logfiles to be stored
```

The next thing I need to specify is the maximum number of logfiles that will be retained. Every time my job runs, it creates output. That output is saved to a logfile that we take a look at in a bit. If the job runs every minute, for the next year, the job would produce a half-million log entries. The chances are that I'd want to see only the last 100 or so; thus, I'll specify `max-log-files 100`, after which the switch will only save the last 100 log entries. The range of acceptable values is inclusive from 1 to 10,000:

```
Arista(config)#schedule Colleen interval 5 timeout 4 max-log-
files ?
<1-10000>  Number of logfiles to be stored
```

Finally, we must include the word `command`, followed by the command to be run:

```
Arista(config)#schedule Colleen int 5 time 4 max-log-files 100
command sho ver
! Schedule a command starting in past
Arista(config)#
```

WARNING

Note that the parser does not check the validity of the commands entered with the `schedule`

command. The command `schedule mistake int 1 max 10 command ILikeCake` will be accepted without complaint, but the job will never run successfully, and you'll never see an error message (though there will be indications when you use `show` commands, as you'll see), or get any cake. If you see *No log files are stored in flash* when you show the job, check to see whether your command is entered correctly. There is no indication as to whether the cake is a lie, so don't bother asking.

With the job now entered, nothing obvious happens. That's pretty anticlimactic, but rest assured that good things are happening (assuming that you entered a valid command). To see what jobs have been scheduled on the switch, use the `show schedule summary` command. Note that the word `summary` must be spelled in its entirety, lest it be confused with the name of a scheduled job:

```
Arista(config)#show schedule summary
Maximum concurrent jobs 1
Name          At time  Last Inter\  Max  Logfile Location
Status
                  time  val      log
                  (mins) files
-----
---
colleen        now  04:51    5    100  flash:/schedule/colleen/
Success
tech-support   now  04:23   60    100  flash:/schedule/tech-support/
Success
```

NOTE

Schedules can be entered with capital letters, but the parser will convert them to all lowercase. For a writer who expects to see his daughter's name in all caps, this is frustrating, even maddening, but it's generally harmless.

To see the specifics of a scheduled job, use the `show schedule job-name` command. This shows the command scheduled, the details of the schedule limitations as configured, and information regarding the logfiles generated as a result of this job. As we can see in this example, my job has run five times:

```
Arista(config)#sho schedule Colleen
The last CLI command execution was successful
CLI command "sho ver" is scheduled next at "20:47:13 01/08/2019",
interval is 5 minutes
Timeout is 4 minutes
Maximum of 100 log files will be stored
Verbose logging is off
5 log files currently stored in flash:/schedule/colleen
```

Start time	Size	Filename
-----	-----	-----
Jan 08 2019 20:42	290.0 bytes	Arista_colleen_2019-01-
08.2042.log.gz		08.2042.log.gz
Jan 08 2019 20:37	290.0 bytes	Arista_colleen_2019-01-
08.2037.log.gz		08.2037.log.gz
Jan 08 2019 20:32	290.0 bytes	Arista_colleen_2019-01-
08.2032.log.gz		08.2032.log.gz
Jan 08 2019 20:27	291.0 bytes	Arista_colleen_2019-01-
08.2027.log.gz		08.2027.log.gz
Jan 08 2019 20:25	290.0 bytes	Arista_colleen_2019-01-
08.2025.log.gz		08.2025.log.gz

See how I passive-aggressively insisted on using the capital C in the job name? That works because CLI is (more or less) case-insensitive. Colleen is a proper noun and thus has a capital C, dammit. You know what's worse? The hostname that is injected by default includes the capital A in Arista!

NOTE

I asked the developers about the whole “forcing lowercase” thing, and the answer they gave me was basically, “I don’t remember why we did this but it’s been that way forever.” That’s not my favorite response, so I dug around in the super-secret internal documentation and found this: *Please note that [the job] name is not case sensitive and is converted to lowercase before it is stored.* It would appear that the reason for the lowercase job names is lost to history. The only thing I hate more than a mystery is a historical mystery, but even wearing my best fedora, I couldn’t unearth the answer in time for my publishing deadline. Not even considering the ~~three~~ ~~four~~ many deadlines that I missed!

Back in the day, if I decided to delete my schedule job, I would do so using the `no schedule job-name` command, and being the passive-aggressive type that I am, I would still include the uppercase letter; the command would be taken but nothing would happen. Now, the job name is just crushed...er, converted to lowercase, no matter how I use it, in order to make the command work:

```
Arista(config)#no schedule Colleen
Arista(config)#sho schedule summary
Maximum concurrent jobs 1
Name          At time Last  Inter\ Max   Logfile Location
Status
                  time    val    log
                  (mins)  files
-----
---
tech-support    now    04:23   60    100   flash:/schedule/tech-support/
Success
Arista(config)#
```

When you delete a scheduled job, the logfiles remain, and because these logfiles are on `flash:`, they persist after a reboot.

By the way, if you’re the type who needs to try to create a job titled

“summary,” I like the way you think! I did exactly that in the first edition of this book, and EOS took it. Well, they’re on to me, and they fixed that and ruined my fun:

```
Arista(config)#schedule summary int 5 time 4 max 1 command sho
int e1
! Schedule a command starting in past
% Job name cannot match keyword summary
```

You might have noticed that there is a job in each example named *tech-support*. This job is installed by Arista and runs on all switches by default. Though you can remove it, there’s generally no reason to do so. I’m sure you’re as curious as I was to see what this job is doing, so let’s take a look, because it’s an excellent example of how we can use `schedule`.

First, let’s use the `show schedule tech-support` command:

```
Arista(config)#sho schedule tech-support
The last CLI command execution was successful
CLI command "show tech-support" is scheduled next at "20:55:22
01/08/2019",
interval is 60 minutes
Timeout is 30 minutes
Maximum of 100 log files will be stored
Verbose logging is off
100 log files currently stored in flash:/schedule/tech-
support
```

Start time	Size	Filename
-----	-----	-----
-----	-----	-----
Jan 08 2019 19:53	74.8 KB	Arista_tech-support_2019-01- 08.1953.log.gz
Jan 08 2019 19:11	80.6 KB	Arista_tech-support_2019-01- 08.1911.log.gz
Jan 08 2019 18:11	80.5 KB	Arista_tech-support_2019-01- 08.1811.log.gz
Jan 08 2019 17:11	80.5 KB	Arista_tech-support_2019-01-

```
08.1711.log.gz
Jan 08 2019 16:11      80.5 KB      Arista_tech-support_2019-01-
08.1611.log.gz
Jan 08 2019 15:11      80.6 KB      Arista_tech-support_2019-01-
08.1511.log.gz
```

As you can see in the second line, this scheduled job performs a **show tech-support** command once every hour. Take a look at the line in the preceding output that's highlighted in bold that says there are 100 logfiles currently stored in *flash:/schedule/tech-support*. It then goes on to list them. How can we see what's in these logfiles? There are a couple of ways. First, with CLI. By concatenating the path from the line in bold with one of the filenames from the list, we can see the contents of any of the files using the **more** command:

```
Arista#more flash:/schedule/tech-support/Arista_tech-
support_2019-01-08.1911.log
.gz
----- show version detail -----
Arista DCS-7280SR-48C6-M-F
Hardware version: 21.05
Deviations: D0003570, D0003550
Serial number: SSJ17290598
System MAC address: 2899.3abe.9f92

[-- output massively truncated --]
```

I'd like to point out a very cool thing about what we just did, and that's the fact that we used the **more** command to show the contents of a binary (g-zipped) file. You can't do that in Bash, and this capability is a benefit of using CLI. If you used the **bash more** command, it would likely pollute your screen with control characters, after which you'd need to issue the **reset** command to clear it up. So how might we view the file in Bash? Let's take a look:

To see the logfiles, drop into Bash and change directories to `/mnt/flash/schedule/`:

```
Arista#bash

Arista Networks EOS shell

[admin@Arista ~]$ cd /mnt/flash/schedule/
[admin@Arista schedule]$ ls
colleen  tech-support
```

We can see that there's a directory for my Colleen (I'll never give up!) job and one for the tech-support job. Changing into the tech-support directory lets us see the logs for the tech-support job:

```
[admin@Arista ~]$ cd tech-support
[admin@Arista tech-support]$ ls
Arista_tech-support_2019-01-04.2220.log.gz
Arista_tech-support_2019-01-04.2320.log.gz
Arista_tech-support_2019-01-05.0020.log.gz
Arista_tech-support_2019-01-05.0120.log.gz
Arista_tech-support_2019-01-05.0220.log.gz
Arista_tech-support_2019-01-05.0320.log.gz
Arista_tech-support_2019-01-05.0420.log.gz
[--- output truncated ---]
```

Because the tech-support job is configured to keep the last 100 log entries, there are 100 files in this directory, each with the date and timestamp in the filename. The files are all in the *gzip* format. To view them, use the `zmore` (or `zcat`) Linux command from Bash. You can also use `gunzip`, but that will expand the file on disk, which is a waste of space when we just want to look at the contents for no other reason than to see what's there:

```
[admin@Arista tech-support]$ zmore Arista_tech-support_2019-01-04.2320.log.gz
-----> Arista_tech-support_2019-01-04.2320.log.gz <-----
```

```
----- show version detail -----
```

```
Arista DCS-7280SR-48C6-M-F
Hardware version: 21.05
Deviations: D0003570, D0003550
Serial number: SSJ17290598
System MAC address: 2899.3abe.9f92

Software image version: 4.21.1F
Architecture: i386
Internal build version: 4.21.1F-9887494.4211F
Internal build ID: 1497e24b-a79b-48e7-a876-43061e109b92

Uptime: 0 weeks, 0 days, 1 hours and 5 minutes
Total memory: 32458980 kB
Free memory: 30481448 kB
```

Installed software packages:

Package	Version	Release
Aaa	1.1.2	9854397.4211F
Aaa-cli	1.1.2	9854397.4211F
Aaa-clientlib	1.1.2	9854397.4211F
Aaa-lib	1.1.2	9854397.4211F
Aboot-utils	1.0.4	9854397.4211F
Acl	1.0.2	9854397.4211F
Acl-cli	1.0.2	9854397.4211F
Acl-lib	1.0.2	9854397.4211F
AclAgent	1.0.3	9854397.4211F
AclAgent-cli	1.0.3	9854397.4211F
AclAgent-lib	1.0.3	9854397.4211F
AclSnmp	1.0.1	9854397.4211F
Adt7462	1.0.2	9854397.4211F
Adt7462-lib	1.0.2	9854397.4211F

--More--

This switch (and all other Arista switches out there) has done a **show tech-support** command every hour since the switch booted. There are 100 files in this directory, which represents the state of the switch in hourly intervals for the past 100 hours. That's 4.166 days for the mathematically challenged. Had a problem yesterday and want to see

the status of the switch? It was saved in these files. Think that's cool and would like to see it save the last 1,000 hours? Just go in and resubmit the tech-support job the way you'd like.

Schedule is cool and all, but what if you want to run more than one command? Not only can the `schedule` command run CLI commands, but it can run bash commands through the use of the `bash` CLI command. Here I've created a script in `/mnt/flash` called `GAD`. I used all caps because it makes me feel like I've won my earlier battle and to show that caps can be used as a reference in the command, even if they can't be used in the schedule job name.

The file looks like this:

```
[admin@Arista flash]$ cat GAD
#!/usr/bin/CLI

sho ver | grep version | email -s "Show Version" gad@gad.net
sho int e24 | email -s "Show int e24" gad@gad.net
```

NOTE

For details on configuring and using the `email` command, see [Chapter 22](#).

Next, back in EOS, I create the schedule that calls this script:

```
Arista(config)#schedule dog int 2 time 1 max 10 command bash
/mnt/flash/GAD
```

And now, every minute, I get two emails showing the requested output. Trust me when I say that this gets old quickly:

```
+ N 2665 May 16 7:16pm Arista@gad.net      (943) Show Version
+ N 2666 May 16 7:16pm Arista@gad.net      (2K) Show int e24
+ N 2667 May 16 7:18pm Arista@gad.net      (943) Show Version
+ N 2668 May 16 7:18pm Arista@gad.net      (2K) Show int e24
+ N 2669 May 16 7:20pm Arista@gad.net      (943) Show Version
+ N 2670 May 16 7:20pm Arista@gad.net      (2K) Show int e24
```

Imagine that you're having a problem, and you'd like to see what the running processes are on your switch every five minutes. Maybe you're convinced that you've got a runaway process on your switch and you want to see for yourself. The Linux command `ps -ef r` shows the running processes:

```
[admin@Arista ~]$ ps -ef r
UID      PID  PPID  C STIME TTY      STAT   TIME CMD
root     1454  1453  3 18:02 ?          R      2:42 Sysdb
root     1827  1453  6 18:03 ?          R      4:54 Mdio
admin    4058  3923 13 19:24 pts/2      R+     0:00 ps -ef r
```

I can schedule a job to send me that information by email every five minutes. The `schedule` command that I would use would look like this:

```
Arista(config)#schedule proc int 5 time 3 max-log-files 1
command
  bashps -ef r | email -s "Running Procs" gad@gad.net
```

There in my email, every five minutes, is a list of the running processes on my switch (email and attachment concatenated in the interest of brevity):

```
Date: Fri, 25 Nov 2018 22:17:25
From: Arista@gad.net
To: gad@gad.net
Subject: Running Procs

UID      PID  PPID  C STIME TTY      STAT   TIME CMD
root    4143  4142 14 19:27 ?          R      0:00 ps -ef r
```

```
root      4144  4142 15 19:27 ?          R      0:00 python /usr/bin/email
-s Running Procs gad@gad.net
```

Conclusion

Hopefully you can see now that schedule can be pretty darn useful. Though it might be a bit less impactful now that tools like CloudVision and eAPI are available, it's still a great way for network engineers to schedule commands without having to use another tool and without having to write any code.

Chapter 25. tcpdump and Advanced Mirroring

tcpdump is an open source packet-capture and analyzer tool that's been around since the late 1980s. *tcpdump* is useful because it allows pretty powerful packet capture sessions from the command line. Even better, you can use it from either Bash or the command-line interface (CLI). Let's take a look. First I show you how it works from within Bash, and then I'll show you what it's like from within EOS.

NOTE

tcpdump will capture only packets destined to or sourced from the CPU. It will not capture data-plane traffic because the CPU couldn't possibly keep up with it all. Well, that's the case on most switches. On some Arista switches you *can* actually see front-panel interface traffic with *tcpdump*! See the end of this chapter for how to use Advanced Mirroring.

tcpdump in Linux

If you have Linux experience and already know how to use *tcpdump*, you might feel more at home using it from Bash. Plus, you'll find that sometimes you *need* to use it from Bash. To do so, just drop into Bash, and have at it:

Arista-Z#**bash**

Arista Networks EOS shell

```
[admin@Arista-Z ~]$ tcpdump -h
tcpdump version 4.9.2
libpcap version 1.8.1
OpenSSL 1.0.2k-fips 26 Jan 2017
Usage: tcpdump [-aAbdDefhHIJKlLnNOPpqStuUvxX#] [ -B size ] [ -c count ]
           [ -C file_size ] [ -E algo:secret ] [ -F file ] [ -G
seconds ]           [ -i interface ] [ -j tstamptype ] [ -M secret ] [ --
number ]           [ -Q in|out|inout ]
           [ -r file ] [ -s snaplen ] [ --time-stamp-precision
precision ]           [ --immediate-mode ] [ -T type ] [ --version ] [ -V file
]
           [ -w file ] [ -W filecount ] [ -y datalinktype ]
           [ -z postrotate-command ]
           [ -Z user ] [ -@ file_index ] [ expression ]
*Use -P to print out info from Arista's DCBs (cpudebug
interface
           only)
*Using -P in combination with -X or -x prints out the
DCB's hex
           and/or ascii representations. This is instead of -x or -
X's
           usual behavior.
```

In its simplest form, tcpdump displays packet information for an interface specified by using the **-i** flag. The thing to remember is that tcpdump is a Linux command and, as such, is looking for Linux interface names, not EOS interface names. This gets me every time I use it. I can be stubbornly stupid sometimes, or so my wife tells me:

```
[admin@Arista-Z ~]$ tcpdump -i e10
tcpdump: e10: No such device exists
(SIOCGIFHWADDR: No such device)
```

Oops—that's not right!

```
[admin@Arista-z ~]$ tcpdump -i Ethernet10
tcpdump: Ethernet10: No such device exists
(SIOCGIFHWADDR: No such device)
```

Dammit! I swear I go through this every time I try to use tcpdump from Linux on an Arista switch, thus proving my wife right, which only enrages me more. An easy way to see the interface names from within the Bash shell is by using the **ifconfig** command. Supplying the **-s** flag will provide a summary list of the interfaces with a bunch of counters. Note that this command puts out a lot of information that extends beyond the end of the page, so I've truncated the last two columns. We don't need them for this exercise anyway because we're looking only for a list of interface names:

```
[admin@Arista-Z ~]$ ifconfig -s
Iface      MTU     RX-OK RX-ERR RX-DRP RX-OVR     TX-OK TX-ERR TX-DRP TX-
OVR Flg
cpu      10209      0      0      0 0          0      0      0
0 BMRU
cpudebug 10209      0      0      0 0          0      0      0
0 BMRU
et1      1500    15835      0      0 0          11262      0      0
0 BMRU
et2      1500    10973      0      0 0          11317      0      0
0 BMRU
et3      1500    52599      0      0 0          3511      0      0
0 BMRU
et4      1500     5049      0      0 0          533      0      0
0 BMU
et5      1500     9568      0      0 0          4953      0      0
0 BMRU
et6      1500     5051      0      0 0          533      0      0
0 BMU
et7      1500     9568      0      0 0          4960      0      0
0 BMRU
et8      1500     9375      0      0 0          4957      0      0
0 BMRU
et9      1500      0      0      0 0          0      0      0
0 BMU
[--- output truncated ---]
```

Ah yes! The interfaces are named such that interface *Ethernet10* in EOS is called *et10* in Bash. And remember, you cannot abbreviate

interface names in Linux like you can in EOS.

NOTE

Technically, the interface names are the *CLI shortnames*, in all lowercase. On modular systems, or systems with multi-lane interfaces or line cards, the slashes in the interface names are replaced with underscores. Thus, e1/1/1 would be et1_1_1 in Linux.

Let's try tcpdump again using the proper name format, this time for Ethernet2:

```
[admin@Arista-Z ~]$ tcpdump -i et2
tcpdump: verbose output suppressed, use -v or -vv for full protocol
decode
listening on et2, link-type EN10MB (Ethernet), capture size 262144 bytes
18:09:45.085787 28:99:3a:26:48:3c (oui Arista Networks) >
01:80:c2:00:00:0e
(oui Unknown), ethertype LLDP (0x88cc), length 197: LLDP, length 183:
Arista-B
18:09:50.812770 28:99:3a:26:87:b7 (oui Arista Networks) >
01:80:c2:00:00:0e
(oui Unknown), ethertype LLDP (0x88cc), length 183: LLDP, length 169:
Arista-Z
[--- output truncated ---]
```

Because tcpdump often creates a lot of output, it's usually best to dump it to a file for later enjoyment. With typical Unix style, I redirect the output to the file *GAD.capture* in the */mnt/flash/* directory. Use Ctrl-C to exit the capture:

```
[admin@Arista-Z ~]$ tcpdump -i ma1 > /mnt/flash/GAD.capture
tcpdump: verbose output suppressed, use -v or -vv for full protocol
decode
listening on ma1, link-type EN10MB (Ethernet), capture size 262144 bytes
^C23 packets captured
26 packets received by filter
0 packets dropped by kernel
```

Really, though, it's better to let tcpdump handle the file creation itself using the `-w` flag:

```
[admin@Arista-Z ~]$ tcpdump -w /mnt/flash/GAD.capture -i ma1
tcpdump: listening on ma1, link-type EN10MB (Ethernet), capture size
262144 bytes
^C87 packets captured
90 packets received by filter
0 packets dropped by kernel
```

If you redirected the output as in the first example, you can view the file as you might view any text file, but if you use the `-w` option in tcpdump, it writes a special file format called *pcap* that is readable by tcpdump, Wireshark, and the like. To read this file on your Arista switch, use the `-r` option with tcpdump:

```
[admin@Arista-Z ~]$ tcpdump -r /mnt/flash/GAD.capture
reading from file /mnt/flash/GAD.capture, link-type EN10MB (Ethernet)
18:12:29.196529 28:99:3a:26:87:b4 (oui Arista Networks) >
ac:1f:6b:00:e7:6f
(oui Unknown), ethertype IPv4 (0x0800), length 202: 10.0.0.101.ssh >
10.0.0.100.48780: Flags [P.], seq 2497468195:2497468331, ack 243613431,
win 303, options [nop,nop,TS val 24726508 ecr 3189908744], length 136
18:12:29.196709 ac:1f:6b:00:e7:6f (oui Unknown) > 28:99:3a:26:87:b4 (oui
Arista Networks), ethertype IPv4 (0x0800), length 66: 10.0.0.100.48780 >
10.0.0.101.ssh: Flags [.], ack 136, win 1444, options [nop,nop,TS val
3189908810 ecr 24726508], length 0
[--output truncated--]
```

Now that we're dumping output to a file, we could even grab more information from each packet while capturing. To do so, use the `-v` option.

Finally, a great option for viewing what's in packets is the `-X` option:

```
[admin@Arista-Z ~]$ tcpdump -X -i et2
tcpdump: verbose output suppressed, use -v or -vv for full protocol
decode
```

```

listening on et2, link-type EN10MB (Ethernet), capture size 262144 bytes
18:19:50.817878 28:99:3a:26:87:b7 (oui Arista Networks) >
01:80:c2:00:00:0e
(oui Unknown), ethertype LLDP (0x88cc), length 183: LLDP, length 169:
Arista-Z
    0x0000: 0207 0428 993a 2687 b504 0a05 4574 6865 ...
(.:&.....Ethe
    0x0010: 726e 6574 3206 0200 780a 0841 7269 7374
rnet2...x..Arist
    0x0020: 612d 5a0c 5341 7269 7374 6120 4e65 7477 a-
Z.SArista.Netw
    0x0030: 6f72 6b73 2045 4f53 2076 6572 7369 6f6e
orks.EOS.version
    0x0040: 2034 2e32 312e 3146 2072 756e 6e69 6e67
.4.21.1F.running
    0x0050: 206f 6e20 616e 2041 7269 7374 6120 4e65
.on.an.Arista.Ne
    0x0060: 7477 6f72 6b73 2044 4353 2d37 3238 3053 tworks.DCS-
7280S
    0x0070: 522d 3438 4336 2d4d 0e04 0014 0014 100c R-48C6-
M.....
    0x0080: 0501 c0a8 6401 0200 4c4b 4100 fe06 0080
....d....LKA.....
    0x0090: c201 0000 fe09 0012 0f03 0100 0000 00fe
.....
    0x00a0: 0600 120f 0427 d800 00 .....'...
18:20:13.697731 28:99:3a:26:48:1d (oui Arista Networks) >
28:99:3a:26:87:b5 (oui
Arista Networks), ethertype IPv4 (0x0800), length 85: 10.10.2.2.bgp >
10.10.2.1.42052: Flags [P.], seq 3755846435:3755846454, ack 2210084905,
win 226,
options [nop,nop,TS val 3764703 ecr 24829041], length 19: BGP
    0x0000: 45c0 0047 a808 4000 0106 b8d2 0a0a 0202
E..G..@.....
    0x0010: 0a0a 0201 00b3 a444 dfdd ab23 83bb 3829
.....D...#..8)
    0x0020: 8018 00e2 1db5 0000 0101 080a 0039 71df
.....9q.
    0x0030: 017a dc71 ffff ffff ffff ffff ffff
.z.q.....
    0x0040: ffff ffff 0013 04 ..... .

```

This can be quite useful for troubleshooting, but the output is increased dramatically, which can be a challenge with large data streams.

To wrap up, here is a nice list of tcpdump flags to play with. Enjoy.

-c number

Limit capture to the specified number of packets

-n

Don't resolve hostnames

-vnn

Don't resolve hostnames or port names

-s number

Limit the capture to the number of bytes specified

-S

Print absolute sequence numbers

-x

Show hex content of packets

-X

Show hex and ASCII content of packets

-xx

Same as -x, but include the Ethernet header

I encourage you to read up more on tcpdump. After you master its use, it can be a very powerful tool, especially on an Arista switch. I'm not going to go any deeper into the Linux side of tcpdump right now because this isn't a Unix book. Let's take a look at how these features are implemented in EOS.

tcpdump in EOS

tcpdump in EOS is much simpler than it is from Linux because

sometime after I wrote the first edition of this book, the developers added the ability to use EOS interface names including abbreviations. Hooray! Plus, we don't need to use all those Linux-looking flags like `-i`. Instead, we just use keywords. Here's a list from using the `?` (question mark) character with `tcpdump`:

```
Arista-Z#tcpdump ?
  file          Set the output file
  filecount     Specify the number of output files
  filter         Set the filtering expression
  interface      Select an interface to monitor (default=fabric)
  lookup-names   Enable reverse DNS lookups
  max-file-size  Specify the maximum size of output file
  monitor        Select a monitor session
  packet-count   Limit number of packets to capture
  queue-monitor  Monitor queue length
  size           Set the maximum number of bytes to dump per packet
  verbose        Enable verbose mode
  <cr>
```

As I mentioned earlier, we can now use interface names! In older versions of EOS, we'd get a message like this if we tried (this is right from the first edition of *Arista Warrior*):

```
Arista#tcpdump int e1
tcpdump: e1: No such device exists
(SIOCGIFHWADDR: No such device)
Starting tcpdump process with command:" tcpdump -i e1 "
```

Now we can use the full or abbreviated names, like any other command in EOS:

```
Arista-Z#tcpdump int e2
tcpdump: verbose output suppressed, use -v or -vv for full protocol
decode
listening on et2, link-type EN10MB (Ethernet), capture size 262144 bytes
18:23:13.696964 28:99:3a:26:48:1d > 28:99:3a:26:87:b5, ethertype IPv4
(0x0800),
  length 85: 10.10.2.2.bgp > 10.10.2.1.42052: Flags [P.],
```

```
seq 3755846492:3755846511, ack 2210084981, win 226,
options [nop,nop,TS val 3809703 ecr 24877314], length 19: BGP
18:23:13.697042 28:99:3a:26:87:b5 > 28:99:3a:26:48:1d, ethertype IPv4
(0x0800),
length 66: 10.10.2.1.42052 > 10.10.2.2.bgp: Flags [.], ack 19, win 229,
options
[nop,nop,TS val 24887633 ecr 3809703], length 0
18:23:15.103521 28:99:3a:26:48:3c > 01:80:c2:00:00:0e, ethertype LLDP
(0x88cc),
length 197: LLDP, length 183: Arista-B
18:23:20.819355 28:99:3a:26:87:b7 > 01:80:c2:00:00:0e, ethertype LLDP
(0x88cc),
length 183: LLDP, length 169: Arista-Z
^C
4 packets captured
4 packets received by filter
0 packets dropped by kernel
```

In old versions of EOS, there was some weirdness when specifying file locations. I'm happy to report that this is no longer the case and that errors like this no longer happen:

```
Arista-EOS-4.9#tcpdump int ma1 file flash:EOS.capture
tcpdump: listening on ma1, link-type EN10MB (Ethernet), capture size
65535 bytes^C
21 packets captured
21 packets received by filter
0 packets dropped by kernel
Starting tcpdump process with command:" tcpdump -w flash:EOS.capture -i
ma1 "
```

If you're wondering what's wrong with that, the concept of *flash:* does not exist in Linux, so that capture would not work the way you might expect.

Nowadays, it all works the way you'd expect it to:

```
Arista-Z#tcpdump int ma1 file flash:EOS.capture
tcpdump: listening on ma1, link-type EN10MB (Ethernet), capture
size 65535 bytes
^C7 packets captured
```

```
8 packets received by filter
0 packets dropped by kernel
```

Be careful, though, because this creates a pcap-formatted file that you must read using `tcpdump -r` because it is not a plain-text file.

NOTE

tcpdump files can be very large, and a lot of testing can litter directories with these files. You should get in the habit of removing tcpdump output files that you no longer need. There's a finite amount of *flash*, and it fills a lot more quickly than most people realize. If your switch has a solid-state drive (SSD), it would be much better to put your files there.

Of course, you can also capture the packets traversing VLAN interfaces. Here, I'll capture packets on the Multichassis Link Aggregation (MLAG) peer-link interface (VLAN 4094):

```
Arista#tcpdump int vlan4094
tcpdump: verbose output suppressed, use -v or -vv for full protocol
decode
listening on vlan4094, link-type EN10MB (Ethernet), capture size 65535
bytes
22:44:27.554755 00:1c:73:17:4a:8e (oui Arista Networks) >
00:1c:73:17:5d:a2 (oui Arista Networks), ethertype IPv4 (0x0800),
length 97: 10.0.0.2.4432 > 10.0.0.1.4432: UDP, length 55
22:44:27.554784 00:1c:73:17:4a:8e (oui Arista Networks) >
00:1c:73:17:5d:a2 (oui Arista Networks), ethertype IPv4 (0x0800),
length 99: 10.0.0.2.4432 > 10.0.0.1.34467: Flags [P.], seq
3108183801:3108183834, ack 4279832075, win 89, options [nop,nop,TS val
42882629 ecr 38856952], length 33
22:44:27.554889 00:1c:73:17:5d:a2 (oui Arista Networks) >
00:1c:73:17:4a:8e (oui Arista Networks), ethertype IPv4 (0x0800),
length 66: 10.0.0.1.34467 > 10.0.0.2.4432: Flags [.], ack 33, win 501,
options [nop,nop,TS val 38857453 ecr 42882629], length 0
[--- output truncated ---]
```

Advanced Mirroring

tcpdump is a great tool for troubleshooting, but the problem that you'll quickly discover is that you can't see most of the traffic you're probably looking for. Using the regular mirror commands available on any Arista switch, the destinations available are other Ethernet interfaces and possibly port-channels and tunnel interfaces, depending on your hardware and EOS version:

```
Arista-7010T(config)#monitor session GAD source ethernet 1
Arista-7010T(config)#monitor session GAD destination ?
  Ethernet      hardware Ethernet interface
  Port-Channel  Lag interface
  tunnel        tunnel keyword
```

Tunnel interfaces allow you to forward mirrored traffic over a Generic Routing Encapsulation (GRE) tunnel so that it can be read somewhere remotely.

Back to Ethernet, the problem is that when you mirror from one interface to another, all of that mirroring is handled by the Application-Specific Integrated Circuit (ASIC), and, because of that, you will not be able to see that traffic with tcpdump.

On some boxes like the 7280R (and others), there is an additional destination available called *Cpu*.

```
Arista-Z(config)#monitor session GAD source e2
Arista-Z(config)#monitor session GAD destination ?
  Cpu        Cpu port(s)
  Ethernet   hardware Ethernet interface
  Port-Channel Lag interface
  tunnel     tunnel keyword
```

Having the ability to forward to the Cpu interface allows us to be able to see our mirrored ports with tcpdump. Hooray! Let me show you. First, Figure 25-1 shows the test lab I've built with three 7280Rs connected.

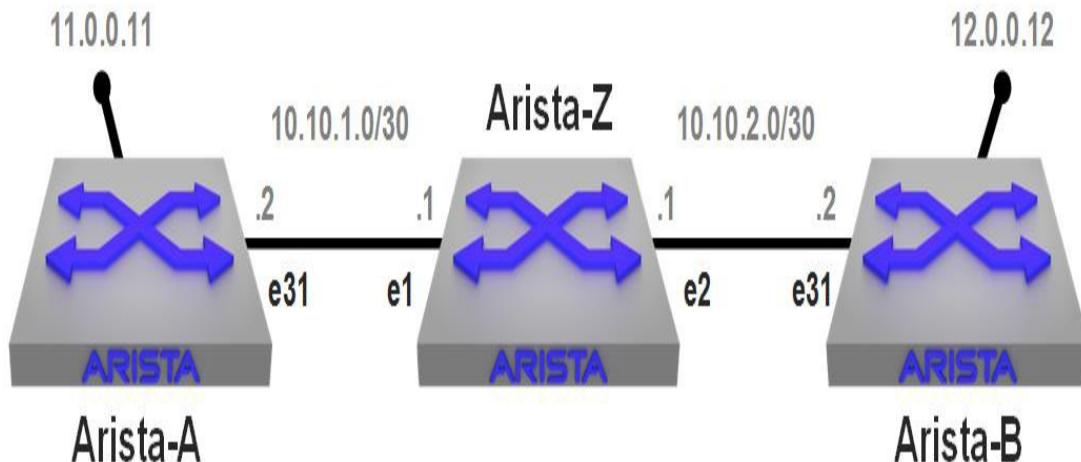


Figure 25-1. Simple lab set up for Advanced Mirroring

I have two switches, Arista-A and Arista-B, separated by Layer 3 (L3) connections, with Arista-Z sitting in the middle doing the routing.

What I'm going to do is set up a continuous ping from Arista-A to the loopback on Arista-B. To show you that the network is as I described, here is a **traceroute** from A to B and then from B to A. I've added IP hostnames in the configurations just to make it all prettier. Here's the route from A to B:

```
Arista-A#traceroute 12.0.0.12
traceroute to 12.0.0.12 (12.0.0.12), 30 hops max, 60 byte packets
 1 Arista-Z (10.10.1.1)  0.304 ms  0.288 ms  0.292 ms
 2 Arista-B (12.0.0.12)  0.230 ms  0.301 ms  0.278 ms
```

And here's the route from B to A:

```
Arista-B#traceroute 11.0.0.11
```

```
traceroute to 11.0.0.11 (11.0.0.11), 30 hops max, 60 byte packets
 1 Arista-Z (10.10.2.1)  0.272 ms  0.228 ms  0.311 ms
 2 Arista-A (11.0.0.11)  0.234 ms  0.359 ms  0.336 ms
```

Now that you can see that there are no shenanigans, I'm going to set up a continuous ping from Arista-A to the loopback on Arista-B. I'm using Bash because it pings continuously by default, whereas the CLI sends only five packets unless I specify otherwise:

```
Arista-A#bash ping 12.0.0.12
PING 12.0.0.12 (12.0.0.12) 56(84) bytes of data.
64 bytes from 12.0.0.12: icmp_seq=1 ttl=63 time=0.269 ms
64 bytes from 12.0.0.12: icmp_seq=2 ttl=63 time=0.256 ms
64 bytes from 12.0.0.12: icmp_seq=3 ttl=63 time=0.246 ms
64 bytes from 12.0.0.12: icmp_seq=4 ttl=63 time=0.254 ms
64 bytes from 12.0.0.12: icmp_seq=5 ttl=63 time=0.245 ms
64 bytes from 12.0.0.12: icmp_seq=6 ttl=63 time=0.252 ms
64 bytes from 12.0.0.12: icmp_seq=7 ttl=63 time=0.244 ms
[--continuous output not shown because it's a book not a video--]
```

Now, from Arista-Z the switch in the middle, I'm going to do a tcpdump of Ethernet 2. Though you can't see it here, the long seconds of no output is annoying. Figure 25-2 depicts the flow of this traffic.

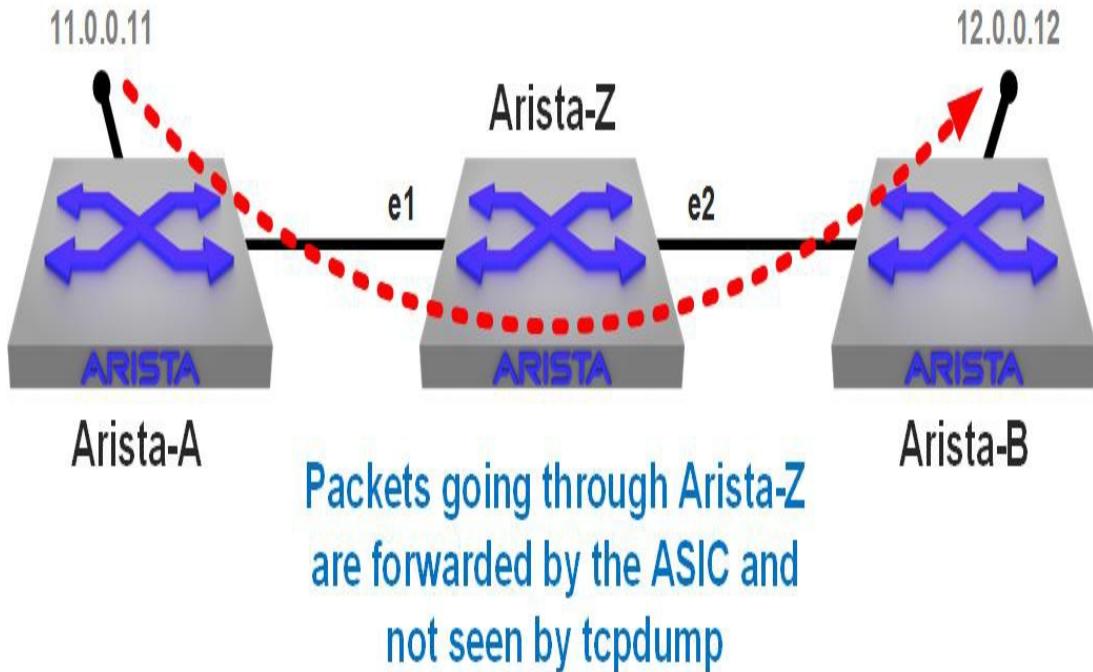


Figure 25-2. Traffic flowing through Arista-Z will not be seen by tcpdump

```
[admin@Arista-Z ~]$ tcpdump -i et2
tcpdump: verbose output suppressed, use -v or -vv for full protocol
decode
listening on et2, link-type EN10MB (Ethernet), capture size 262144 bytes
18:59:45.133915 28:99:3a:26:48:3c (oui Arista Networks) >
01:80:c2:00:00:0e
(oui Unknown), ethertype LLDP (0x88cc), length 197: LLDP, length 183:
Arista-B
18:59:50.866383 28:99:3a:26:87:b7 (oui Arista Networks) >
01:80:c2:00:00:0e
(oui Unknown), ethertype LLDP (0x88cc), length 183: LLDP, length 169:
Arista-Z
18:59:51.831411 28:99:3a:26:87:b5 (oui Arista Networks) >
28:99:3a:26:48:1d
(oui Arista Networks), ethertype IPv4 (0x0800), length 85:
10.10.2.1.42052
10.10.2.2.bgp: Flags [P.], seq 2210085779:2210085798, ack 3755847195, win
229, options [nop,nop,TS val 25437167 ecr 4349703], length 19: BGP
18:59:51.831626 28:99:3a:26:48:1d (oui Arista Networks) >
28:99:3a:26:87:b5
(oui Arista Networks), ethertype IPv4 (0x0800), length 66:
10.10.2.2.bgp >
10.10.2.1.42052: Flags [.], ack 19, win 226, options [nop,nop,TS val
4359239
```

```

ecr 25437167], length 0
19:00:13.687100 28:99:3a:26:48:1d (oui Arista Networks) >
28:99:3a:26:87:b5
(oui Arista Networks), ethertype IPv4 (0x0800), length 85:
10.10.2.2.bgp >
10.10.2.1.42052: Flags [P.], seq 1:20, ack 19, win 226, options
[nop,nop,TS
val 4364703 ecr 25437167], length 19: BGP
19:00:13.687173 28:99:3a:26:87:b5 (oui Arista Networks) >
28:99:3a:26:48:1d
(oui Arista Networks), ethertype IPv4 (0x0800), length 66:
10.10.2.1.42052
10.10.2.2.bgp: Flags [.], ack 20, win 229, options [nop,nop,TS val
25442631
ecr 4364703], length 0
19:00:15.133815 28:99:3a:26:48:3c (oui Arista Networks) >
01:80:c2:00:00:0e
(oui Unknown), ethertype LLDP (0x88cc), length 197: LLDP, length 183:
Arista-B
19:00:20.867199 28:99:3a:26:87:b7 (oui Arista Networks) >
01:80:c2:00:00:0e
(oui Unknown), ethertype LLDP (0x88cc), length 183: LLDP, length 169:
Arista-Z
19:00:45.133717 28:99:3a:26:48:3c (oui Arista Networks) >
01:80:c2:00:00:0e
(oui Unknown), ethertype LLDP (0x88cc), length 197: LLDP, length 183:
Arista-B
^C
9 packets captured
9 packets received by filter
0 packets dropped by kernel

```

Not a single ping packet, and by the time I'd captured that output, 340 had been sent. Again, this is because tcpdump sees only packets sourced from or destined to the switch itself. To prove that, here I ping the Ethernet 2 IP address on Arista-Z from Arista-B (the directly connected switch on the right of the drawing):

```

[admin@Arista-Z ~]$ tcpdump -i et2
tcpdump: verbose output suppressed, use -v or -vv for full protocol
decode
listening on et2, link-type EN10MB (Ethernet), capture size 262144 bytes
19:07:25.950323 28:99:3a:26:48:1d (oui Arista Networks) >

```

```

28:99:3a:26:87:b5
(oui Arista Networks), ethertype IPv4 (0x0800), length 114: 10.10.2.2 >
10.10.2.1: ICMP echo request, id 596, seq 1, length 80
19:07:25.950393 28:99:3a:26:87:b5 (oui Arista Networks) >
28:99:3a:26:48:1d
(oui Arista Networks), ethertype IPv4 (0x0800), length 114: 10.10.2.1 >
10.10.2.2: ICMP echo reply, id 596, seq 1, length 80
19:07:25.950578 28:99:3a:26:48:1d (oui Arista Networks) >
28:99:3a:26:87:b5
(oui Arista Networks), ethertype IPv4 (0x0800), length 114: 10.10.2.2 >
10.10.2.1: ICMP echo request, id 596, seq 2, length 80
19:07:25.950595 28:99:3a:26:87:b5 (oui Arista Networks) >
28:99:3a:26:48:1d
(oui Arista Networks), ethertype IPv4 (0x0800), length 114: 10.10.2.1 >
10.10.2.2: ICMP echo reply, id 596, seq 2, length 80
19:07:25.950737 28:99:3a:26:48:1d (oui Arista Networks) >
28:99:3a:26:87:b5
(oui Arista Networks), ethertype IPv4 (0x0800), length 114: 10.10.2.2 >
10.10.2.1: ICMP echo request, id 596, seq 3, length 80
19:07:25.950759 28:99:3a:26:87:b5 (oui Arista Networks) >
28:99:3a:26:48:1d
(oui Arista Networks), ethertype IPv4 (0x0800), length 114: 10.10.2.1 >
10.10.2.2: ICMP echo reply, id 596, seq 3, length 80
19:07:25.950896 28:99:3a:26:48:1d (oui Arista Networks) >
28:99:3a:26:87:b5
(oui Arista Networks), ethertype IPv4 (0x0800), length 114: 10.10.2.2 >
10.10.2.1: ICMP echo request, id 596, seq 4, length 80
^C
7 packets captured
10 packets received by filter
3 packets dropped by kernel
[admin@Arista-Z ~]$

```

There we go. Packets sent to the switch (and the replies from the switch) are seen by tcpdump, but packets going through the switch are not.

Now let's turn on the Advanced Mirroring capability and mirror to the CPU:

```

Arista-Z(config)#monitor session GAD source ethernet 1 rx
Arista-Z(config)#monitor session GAD source ethernet 2 rx

```

```
Arista-Z(config)#monitor session GAD destination cpu
```

The trick now is to determine where to point tcpdump. To do that, use the `show monitor` command:

```
Arista-Z(config)#sho monitor session GAD
```

```
Session GAD
```

```
-----
```

```
Source Ports:
```

```
Rx Only: Et1, Et2
```

```
Destination Ports:
```

```
Cpu : active (mirror0)
```

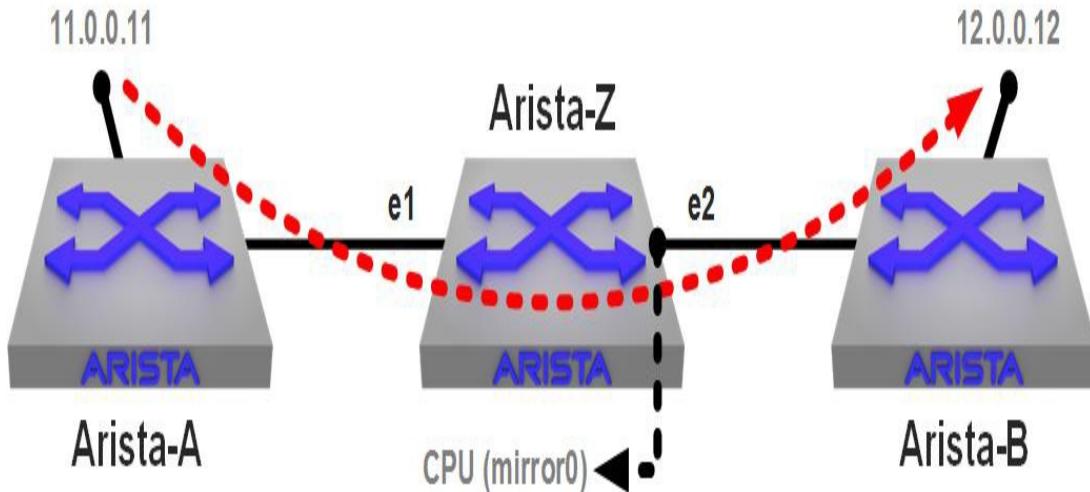
On that last line it shows a new interface name in parentheses called `mirror0`. It's `mirror0` because it's the first monitor session I've created since booting the switch. If I create another, it will be assigned to `mirror1`, and so on. On this switch, there are 16 available mirror interfaces (you can see them by using the Bash command `ifconfig -s | grep mirror`).

Now that I have a `mirror0` interface, I can point tcpdump there like any other interface. Here, I am doing just that while also limiting the capture to five packets:

```
[admin@Arista-Z ~]$ tcpdump -i mirror0 -c5
tcpdump: verbose output suppressed, use -v or -vv for full protocol
decode
listening on mirror0, link-type EN10MB (Ethernet), capture size 262144
bytes
19:16:58.916211 28:99:3a:18:46:f1 (oui Arista Networks) >
28:99:3a:26:87:b5
(oui Arista Networks), ethertype IPv4 (0x0800), length 98: 10.10.1.2 >
12.0.0.12: ICMP echo request, id 7345, seq 1017, length 64
```

```
19:16:58.916219 28:99:3a:26:87:b5 (oui Arista Networks) >
28:99:3a:26:48:1d
(oui Arista Networks), ethertype IPv4 (0x0800), length 98: 10.10.1.2 >
12.0.0.12: ICMP echo request, id 7345, seq 1017, length 64
19:16:58.916319 28:99:3a:26:48:1d (oui Arista Networks) >
28:99:3a:26:87:b5
(oui Arista Networks), ethertype IPv4 (0x0800), length 98: 12.0.0.12 >
10.10.1.2: ICMP echo reply, id 7345, seq 1017, length 64
19:16:58.916322 28:99:3a:26:87:b5 (oui Arista Networks) >
28:99:3a:18:46:f1
(oui Arista Networks), ethertype IPv4 (0x0800), length 98: 12.0.0.12 >
10.10.1.2: ICMP echo reply, id 7345, seq 1017, length 64
19:16:59.915884 28:99:3a:18:46:f1 (oui Arista Networks) >
28:99:3a:26:87:b5
(oui Arista Networks), ethertype IPv4 (0x0800), length 98: 10.10.1.2 >
12.0.0.12: ICMP echo request, id 7345, seq 1018, length 64
5 packets captured
8 packets received by filter
0 packets dropped by kernel
```

I've highlighted the source and destination IP addresses and the packet types. They're all Internet Control Message Protocol (ICMP) packets and the source and destination IP addresses are not on this switch, as shown in [Figure 25-3](#).



With Advanced Mirroring,
 Packets can be mirrored to the
 CPU so they can be seen by
`tcpdump`

Figure 25-3. *Tcpdump with Advanced Mirroring*

Nice! We can now mirror anything right to the CPU, right? Well, not exactly. You see, EOS protects the CPU from being overwhelmed by something called a CoPP filter. CoPP stands for Control Plane Policing, and it limits how much can be sent to the CPU. On older revisions of code, that limit was about 400 Mbps. Now it's about 800 Mbps. That's not a lot considering the fact that I'm mirroring a 10 Gbps interface, and what about switches that have 100 Gbps or 400 Gbps interfaces?

Filtering Advanced Mirroring Sessions

Luckily, there are a couple of tools available to help you with the

limitation imposed by CoPP. The first is the ability to apply an access list to the monitor session, which will then mirror only what matches the Access Control List (ACL). I really dig this because I think of it as if I'm applying a filter on a WAN link: why filter after it crosses the link? Similarly, why collect all the data just to filter it later? I feel like I've been doing it wrong for decades, but I've never seen a switch that could filter before the mirroring was applied. Gotta love Arista.

To do this, begin by making an ACL. I'm going to make a super-simple one that permits any traffic to or from 12.0.0.12 (the loopback on Arista-B) and ignores everything else:

```
Arista-Z(config)#ip access-list Filter12
Arista-Z(config-acl-GAD)#10 permit ip host 12.0.0.12 any
Arista-Z(config-acl-GAD)#20 permit ip any host 12.0.0.12
Arista-Z(config-acl-GAD)#exit
Arista-Z(config)#
```

Now I can apply it to my monitor session like so:

```
Arista-Z(config)#monitor session GAD ip access-group Filter12
```

To see whether it's applied, use the `show monitor session` command:

```
Arista-Z#sho monitor sess GAD

Session GAD
-----
Source Ports:
Rx Only:      Et1(IP ACL: Filter12), Et2(IP ACL: Filter12)

Destination Ports:
```

```
Cpu : active (mirror0)  
ip access-group: Filter12
```

If you have multiple source interfaces, you can apply the ACL to individual interfaces or all (the default).

NOTE

There is a limitation of the ASIC used in the 7280R that results in ACLs applied to the TX direction not working, so that's why my monitor session is configured to use RX on each of the interfaces involved.

To make things more interesting, I'm now going to set up a massive amount of traffic using iperf. If you weren't aware, iperf is now part of the standard EOS build. I'm going to set up the client on Arista-A, and the server on Arista-B.

Before we can do that, though, we need to allow the switch to listen on TCP port 5001. To make that happen, I update the control-plane **access-list** to allow that to work. Here's my updated ACL:

```
ip access-list Allow-Iperf  
statistics per-entry  
10 permit icmp any any  
20 permit ip any any tracked  
30 permit udp any any eq bfd ttl eq 255  
40 permit udp any any eq bfd-echo ttl eq 254  
50 permit udp any any eq multihop-bfd  
60 permit udp any any eq micro-bfd  
70 permit ospf any any  
80 permit tcp any any eq 5001 ssh telnet www snmp bgp https msdp ldp  
90 permit udp any any eq bootps bootpc snmp rip ntp ldp  
100 permit tcp any any eq mlag ttl eq 255  
110 permit udp any any eq mlag ttl eq 255  
120 permit vrrp any any
```

```
130 permit ahp any any
140 permit pim any any
150 permit igmp any any
160 permit tcp any any range 5900 5910
170 permit tcp any any range 50000 50100
180 permit udp any any range 51000 51100
190 permit tcp any any eq 3333
200 permit tcp any any eq nat ttl eq 255
210 permit tcp any eq bgp any
220 permit rsvp any any
```

You need to apply that to the control-plane, like so:

```
Arista-B(config)#control-plane
Arista-B(config-cp)#ip access-group Allow-Iperf in
```

NOTE

From about EOS 4.16 to EOS 4.20, changing access to the switch can be a very different process. This is the standard as of EOS 4.21 and on EOS 4.15 and prior.

OK, let's set up the iperf server on Arista-B. You do this from Bash:

```
[admin@Arista-B ~]$ iperf -s
-----
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
-----
```

That now sits and waits, and we can now hit it with an iperf client on Arista-A. This is a new Secure Shell (SSH) session because my original one is going to be used for sending pings. I'm going to limit my pings to one every five seconds and send only three to simplify the output. I'm sending only three because iperf runs for 10 seconds, and that way I don't need to break out of the pings.

Note that I'm going to hit the Ethernet interface on Arista-A and not the loopback that we've been using. This results in the iperf packets still being forwarded by Arista-Z's ASIC, but they will not be matched by the ACL that I made on the monitor session, because of the different destination IP.

At the same time that I start the iperf onslaught, I'm going to start the tcpdump on Arista-Z. As soon as the iperf is done, I'll stop the capture and show all four windows' output.

Here is the ping that took place on Arista-A:

```
[admin@Arista-A ~]$ ping -i 5 -c 3 12.0.0.12
PING 12.0.0.12 (12.0.0.12) 56(84) bytes of data.
64 bytes from 12.0.0.12: icmp_seq=1 ttl=63 time=7.45 ms
64 bytes from 12.0.0.12: icmp_seq=2 ttl=63 time=6.01 ms
64 bytes from 12.0.0.12: icmp_seq=3 ttl=63 time=0.247 ms

--- 12.0.0.12 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 10008ms
rtt min/avg/max/mdev = 0.247/4.573/7.458/3.116 ms
[admin@Arista-A ~]$
```

Here is the iperf in action from Arista-A's perspective:

```
[admin@Arista-A ~]$ iperf -c 10.10.2.2
-----
Client connecting to 10.10.2.2, TCP port 5001
TCP window size: 45.0 KByte (default)
-----
[  3] local 10.10.1.2 port 44154 connected with 10.10.2.2 port 5001
[ ID] Interval      Transfer     Bandwidth
[  3]  0.0-10.0 sec   144 MBytes   120 Mbits/sec
[admin@Arista-A ~]$
```

In the 10 seconds that this test ran, iperf sent 144 MB of traffic from Arista-A, through Arista-Z, and to the Ethernet IP address of Arista-B.

Here's what Arista B saw regarding iperf:

```
[admin@Arista-B ~]$ iperf -s
-----
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
-----
[ 4] local 10.10.2.2 port 5001 connected with 10.10.1.2 port 44154
[ ID] Interval      Transfer     Bandwidth
[ 4]  0.0-10.0 sec  144 MBytes   120 Mbits/sec
```

Finally, here's the result of the tcpdump on Arista-Z. What should we see? Well, there's 144 MB of traffic squirting through these interfaces being mirrored, but remember that there's an ACL on the monitor session, so we should see only the pings sourced from or destined to 12.0.0.12, which is the host specified in the ACL. Let's take a look:

```
[admin@Arista-Z ~]$ tcpdump -i mirror0
tcpdump: verbose output suppressed, use -v or -vv for full protocol
decode
listening on mirror0, link-type EN10MB (Ethernet), capture size 262144
bytes
10:58:25.825402 28:99:3a:18:46:f1 (oui Arista Networks) >
28:99:3a:26:87:b5
(oui Arista Networks), ethertype IPv4 (0x0800), length 98: 10.10.1.2 >
12.0.0.12: ICMP echo request, id 2094, seq 1, length 64
10:58:25.832740 28:99:3a:26:48:1d (oui Arista Networks) >
28:99:3a:26:87:b5
(oui Arista Networks), ethertype IPv4 (0x0800), length 98: 12.0.0.12 >
10.10.1.2: ICMP echo reply, id 2094, seq 1, length 64
10:58:30.829619 28:99:3a:18:46:f1 (oui Arista Networks) >
28:99:3a:26:87:b5
(oui Arista Networks), ethertype IPv4 (0x0800), length 98: 10.10.1.2 >
12.0.0.12: ICMP echo request, id 2094, seq 2, length 64
10:58:30.835558 28:99:3a:26:48:1d (oui Arista Networks) >
28:99:3a:26:87:b5
(oui Arista Networks), ethertype IPv4 (0x0800), length 98: 12.0.0.12 >
10.10.1.2: ICMP echo reply, id 2094, seq 2, length 64
10:58:35.833722 28:99:3a:18:46:f1 (oui Arista Networks) >
28:99:3a:26:87:b5
(oui Arista Networks), ethertype IPv4 (0x0800), length 98: 10.10.1.2 >
12.0.0.12: ICMP echo request, id 2094, seq 3, length 64
```

```
10:58:35.8333821 28:99:3a:26:48:1d (oui Arista Networks) >
28:99:3a:26:87:b5
(oui Arista Networks), ethertype IPv4 (0x0800), length 98: 12.0.0.12 >
10.10.1.2: ICMP echo reply, id 2094, seq 3, length 64
^C
6 packets captured
6 packets received by filter
0 packets dropped by kernel
```

Pretty cool, huh? With Advanced Mirroring, gone are the days of capturing 144 MB only to have to filter that output after the capture. Now we can filter at the source! That certainly helps with our CoPP limit, but there's another trick we can do, and that's truncation.

Truncation with Advanced Mirroring

As network engineers, we're usually not interested in the payload of the packets and are instead usually examining things like TCP window sizes, User Datagram Protocol (UDP) port numbers, ACKs, and the like. If you're capturing, for example, an iSCSI stream, that can be a huge amount of data that ends up being forwarded to the CPU with Advanced Mirroring only to be throttled by CoPP. Advanced Mirroring allows you to truncate the mirrored packets, and it's quite easy to do:

```
Arista-Z(config)#monitor session GAD truncate
```

Depending on your hardware and EOS version, you might be able to set size options as well. On my 7280R, I have the following options:

```
Arista-Z(config)#monitor session GAD truncate size ?
128  Nominal mirroring truncation size in bytes
192  Nominal mirroring truncation size in bytes
```

I choose the bigger one because bigger is always better. Well, not

always, but I grew up in the 70s when that was a fundamental truth in life, so bigger it is.

```
Arista-Z(config)#monitor session GAD truncate size 192
```

This time I'm going to fire off my three pings, but I'm going to make them 1,000 bytes apiece:

```
[admin@Arista-A ~]$ ping -i 5 -c 3 -s 1000 12.0.0.12
PING 12.0.0.12 (12.0.0.12) 1000(1028) bytes of data.
1008 bytes from 12.0.0.12: icmp_seq=1 ttl=63 time=0.272 ms
1008 bytes from 12.0.0.12: icmp_seq=2 ttl=63 time=0.275 ms
1008 bytes from 12.0.0.12: icmp_seq=3 ttl=63 time=0.274 ms

--- 12.0.0.12 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 10000ms
rtt min/avg/max/mdev = 0.272/0.273/0.275/0.019 ms
```

Let's see what Arista-Z's mirror session thinks of that:

```
[[admin@Arista-Z ~]$ tcpdump -i mirror0
tcpdump: verbose output suppressed, use -v or -vv for full protocol
decode
listening on mirror0, link-type EN10MB (Ethernet), capture size 262144
bytes
11:20:41.740608 28:99:3a:18:46:f1 (oui Arista Networks) >
28:99:3a:26:87:b5
(oui Arista Networks), ethertype IPv4 (0x0800), length 154: truncated-
ip -
  888 bytes missing! 10.10.1.2 > 12.0.0.12: ICMP echo request, id 4480,
seq 1,
  length 1008
11:20:41.740711 28:99:3a:26:48:1d (oui Arista Networks) >
28:99:3a:26:87:b5
(oui Arista Networks), ethertype IPv4 (0x0800), length 154: truncated-
ip -
  888 bytes missing! 12.0.0.12 > 10.10.1.2: ICMP echo reply, id 4480,
seq 1,
  length 1008
11:20:46.742403 28:99:3a:18:46:f1 (oui Arista Networks) >
```

```

28:99:3a:26:87:b5
(oui Arista Networks), ethertype IPv4 (0x0800), length 154: truncated-
ip -
888 bytes missing! 10.10.1.2 > 12.0.0.12: ICMP echo request, id 4480,
seq 2,
length 1008
11:20:46.742514 28:99:3a:26:48:1d (oui Arista Networks) >
28:99:3a:26:87:b5
(oui Arista Networks), ethertype IPv4 (0x0800), length 154: truncated-
ip -
888 bytes missing! 12.0.0.12 > 10.10.1.2: ICMP echo reply, id 4480,
seq 2,
length 1008
11:20:51.741460 28:99:3a:18:46:f1 (oui Arista Networks) >
28:99:3a:26:87:b5
(oui Arista Networks), ethertype IPv4 (0x0800), length 154: truncated-
ip -
888 bytes missing! 10.10.1.2 > 12.0.0.12: ICMP echo request, id 4480,
seq 3,
length 1008
11:20:51.741569 28:99:3a:26:48:1d (oui Arista Networks) >
28:99:3a:26:87:b5
(oui Arista Networks), ethertype IPv4 (0x0800), length 154: truncated-
ip -
888 bytes missing! 12.0.0.12 > 10.10.1.2: ICMP echo reply, id 4480,
seq 3,
length 1008
^C
6 packets captured
6 packets received by filter
0 packets dropped by kernel
[admin@Arista-Z ~]$
```

We've now filtered the traffic that was being mirrored and also truncated the packets. If this were a large stream of data, we could effectively mirror only the header information, which would therefore (probably) stay well within the CoPP limitation.

Conclusion

tcpdump and especially Advanced Mirroring are powerful

troubleshooting tools, but you need to exercise care when mirroring to the CPU because it can have an impact. That's why there's a CoPP filter in the first place. I still greatly prefer to have a real sniffer, even if it's Wireshark on my laptop, but in an emergency, this can be a great tool to help you when other tools might not be readily available or even possible. How might a tool not be possible? I can think of a pile of times in my past when I would have given big money to have been able to use tcpdump like this on a remote box at 4 a.m. during an outage. With the right Arista box, that's now a reality.

Chapter 26. Tap Aggregation

Often called “TapAgg” by the Arista team, this feature is the direct result of the amazing hardware design of Arista switches coupled with some brilliant software engineering.

NOTE

Fan-boy much? Read on, and if you don’t agree that this is great, I’ll buy you lunch. Actually, that’s not true; I won’t buy you lunch. I’ll buy myself lunch, though, and raise a toast in your name. Actually, I don’t really drink, so I won’t do that either.

Tap aggregation is the ability to connect multiple devices to a sniffer or analyzer or even groups of tools. Because good analyzers are generally very expensive and most important network devices reside in remote data centers, the ability to aggregate and control the connections to the devices has spawned an industry of tap aggregation devices. These devices tend to also be very expensive, but in the remote data centers, they can be a very valuable addition to the troubleshooting toolset of the networking team. If you’ve ever struggled with the problem of needing to monitor the traffic on one switch while the dedicated sniffer is attached to another switch—thus resulting in you driving to the data center to move a cable—tap aggregation is for you.

Arista primarily sells switches (though it has broadened what switches can do), so why is it involved in the realm of tap aggregation? Arista

switches weren't designed to be tap aggregation devices, but somewhere along the lines, someone discovered that because the switches are completely nonblocking, they can be easily configured to perform the same function as a tap aggregation device. Not only that, but they can do the same job as an expensive tap aggregation device for a fraction of the cost. Although tap aggregators tend to be expensive, tap aggregators capable of delivering 10 Gbps ports are often prohibitively so.

WARNING

A quick warning is probably warranted. This feature essentially disables normal switching on the device. Do not experiment with this feature on a production switch! Doing so will likely result in bad things happening, such as the termination of your employment.

Let's configure a 7280R for tap aggregation, which should explain some of the details for this feature. First, we need to put the switch into tap aggregation mode:

```
Arista(config)#tap aggregation
Arista(config-tap-agg)#mode exclusive
```

The only mode available on this switch is **exclusive**. On a 7500R or other tap aggregation–capable chassis switch, you can enable hybrid or **mixed** mode and enable certain blades to be tap aggregation devices while leaving the other blades to continue operating as normal:

```
Arista-7508R(config)#tap aggregation
Arista-7508R(config-tap-agg)#mode mixed module linecard 3-4, 6
```

When this is done, assuming no preexisting tap aggregation configuration exists, all of the ports on the switch (or linecard) are placed into **errdisabled** mode. Back on our 7280R, here's what the interfaces look like:

```
Arista#sho int status
Port      Name    Status      Vlan      Duplex  Speed  Type
Et1       Et1     errdisabled  1          full    10G   10GBASE-SRL
Et2       Et2     errdisabled  1          full    10G   10GBASE-SRL
Et3       Et3     errdisabled  1          full    10G   10GBASE-SRL
Et4       Et4     errdisabled  1          full    10G   10GBASE-SR
Et5       Et5     errdisabled  1          full    10G   Not Present
Et6       Et6     errdisabled  1          full    10G   Not Present
Et7       Et7     errdisabled  1          a-full  a-10G Not Present
Et8       Et8     errdisabled  1          full    10G   Not Present
[-- snip --]
```

To disable tap-aggregation mode, simply negate the tap aggregation command:

```
Arista(config)#no tap aggregation
```

We're not going to do that now, though, because we're trying to learn about this feature and disabling it would be counterproductive to that end.

When the switch is put into tap aggregation mode, it essentially stops functioning as a switch. Specifically, CPU-generated packets such as Spanning Tree Protocol (STP), Link Layer Discovery Protocol (LLDP), and the like are no longer sent to tool ports. Features such as Internet Group Management Protocol (IGMP) snooping and LLDP are prevented from running on tap or tool ports, though they remain running because future code revisions might allow hybrid mode. Arista recommends disabling these protocols when running in tap aggregation

mode. Features such as Access Control Lists (ACLs), quality of service (QoS), and Link Aggregation Groups (LAGs) remain enabled since they may be of use with tap aggregation.

Ports on the switch can now be placed into one of two modes: *tap* or *tool*:

tap

A tap port is a port connected to either a tap device or a monitor port on a switch (or any other device delivering packets destined for a sniffer or analyzer). A tap port receives traffic.

tool

A tool port is one to which a sniffer or analyzer is usually connected. This is not always the case, as the destination may be another tap aggregation device, as you'll see later in this chapter. A tool port sends traffic.

Tap and tool ports are associated through the use of an *agg-group*. A tool port can belong to multiple agg-groups, but a tap port can belong to only one. This allows a source to be sent to multiple analyzers, if so desired. Tap and tool ports can be physical interfaces or port channels.

To configure a port as a tool port, use the **switchport mode tool** interface command:

```
Arista(config)#int e10
Arista(config-if-Et10)#switchport mode tool
```

To configure a port as a tap port, use the **switchport mode tap** interface command:

```
Arista(config-if-Et10)#int e11
```

```
Arista(config-if-Et11)#switchport mode tap
```

These changes are reflected in the output of `show interface status`:

```
Arista#sho int status
Port      Name      Status      Vlan      Duplex      Speed      Type
Et1        errdisabled 1          full      10G 10GBASE-SRL
Et2        errdisabled 1          full      10G 10GBASE-SRL
Et3        errdisabled 1          full      10G 10GBASE-SRL
Et4        errdisabled 1          full      10G 10GBASE-SR
Et5        errdisabled 1          full      10G Not Present
Et6        errdisabled 1          full      10G Not Present
Et7        errdisabled 1          a-full    a-10G Not Present
Et8        errdisabled 1          full      10G Not Present
Et9        errdisabled 1          full      10G Not Present
Et10       errdisabled tool      full      10G
Not Present
Et11       errdisabled tap      full      10G
Not Present
[-- snip --]
```

NOTE

When interfaces are placed into tap or tool mode, normal Ethernet learning and flooding behavior is disabled. This helps to ensure that the tap ports remain RX-only and the tool ports remain TX-only. Additionally, Layer 2 and Layer 3 protocols are prevented from sending protocol packets to tap and tool ports.

For our tap aggregation ports to be of any use, we need to add them to an aggregation group. This is done a little differently based on the port type configured. To better illustrate this, I've added one more of each type of port. Here is how the ports are configured, as shown by the output of `show interface status` with some *grepping* in the interest of brevity:

```
Arista#sho int status | egrep 'tap|tool'
```

Et10	notconnect	tool	full	10G Not Present
Et11	notconnect	tap	full	10G Not Present
Et12	notconnect	tap	full	10G Not Present
Et13	notconnect	tap	auto	auto 1000BASE-T
Et20	notconnect	tool	full	10G Not Present
Et21	notconnect	tap	full	10G 10GBASE-SRL
Et22	notconnect	tap	full	10G Not Present
Et23	connected	tap	full	10G 10GBASE-SRL

Given this configuration, I would like to have interfaces Et11, 12, and 13 forward their packets to the tool connected to interface Et10, whereas the interfaces Et21, 22, and 23 should forward their packets to the tool at interface Et20. Grammatically, I should probably have written that the interfaces should forward their packets to the tool *connected* to the respective interfaces, but the image of a guy I'd describe as a “tool” sitting in front of whatever device he's using to capture packets makes me giggle. Grouping all of the 1x ports together and all of the 2x ports together tickles my OCD, and humor helps to take the edge off.

Tool ports can be included in multiple groups, whereas tap ports cannot. Let's get the more complicated possibility out of the way first and look at a tool port:

```
Arista(config-if-Et10)#switchport tool group ?
  add      Add groups to the current list
  remove   Remove groups from the current list
  set      Set groups for the interface
```

I'll add two groups, called **Sniffer** and **Analyzer**. Port Et10 will have the **Sniffer**, and E20 will have the **Analyzer**.

```
Arista(config-if-Et10)#switchport tool group set Sniffer
```

Here's the configuration for Et20:

```
Arista(config-if-Et10)#int e20
Arista(config-if-Et20)#switchport tool group set Analyzer
```

Tap ports are configured a little bit differently. To configure ports Et11 through 13 to belong in the **Sniffer** group, I use the **switchport tap** interface command:

```
Arista(config-if-Et11-13)#switchport tap ?
  allowed      Configure allowed vlans when interface is in tap mode
  default      Configure default tap group
  identity     Configure tap port id
  native       set native vlan when interface is in tap mode
  truncation   Configure if ingress packet should be truncated
```

We'll look at the other options in a bit. For now, we use the **default group** option:

```
Arista(config)#int e11-13
Arista(config-if-Et11-13)#switchport tap default group Sniffer
```

Now let's do the same for ports Et21 through 23, placing them into the **Analyzer** group:

```
Arista(config-if-Et11-13)#int e21-23
Arista(config-if-Et21-23)#switchport tap default group Analyzer
```

To see the current status of the tap aggregation groups, use the **show tap aggregation groups** command:

```
Arista#sho tap aggregation groups
Group Name          Tool Members
-----
Sniffer            None
Analyzer          None
```

Group Name	Tap Members
Sniffer	None
Analyzer	Et23

Where are all of the interfaces we just configured? They're not reported here because they're not currently connected. Look back to the `show interface status` command a few paragraphs back. See how only one interface is connected? That's the one that shows up now. This can be a little confusing if you're not familiar with how the feature works, but just remember that this is showing active ports, not configured ports. The command is reporting that there are no tool members for either group because there are no active tool members in either group. Let's see what happens when I connect up ports Et10 through 13:

Arista#sho tap aggregation groups	
Group Name	Tool Members
Sniffer	Et10
Analyzer	None
Group Name	
Tap Members	
Sniffer	Et11, Et12, Et13
Analyzer	Et23

That's a little better! Let's see it in action.

Tap Aggregation from the Command-Line Interface

Figure 26-1 presents the lab scenario that I've built to show how we

can use groups to affect the flow of traffic in a tap aggregation switch.

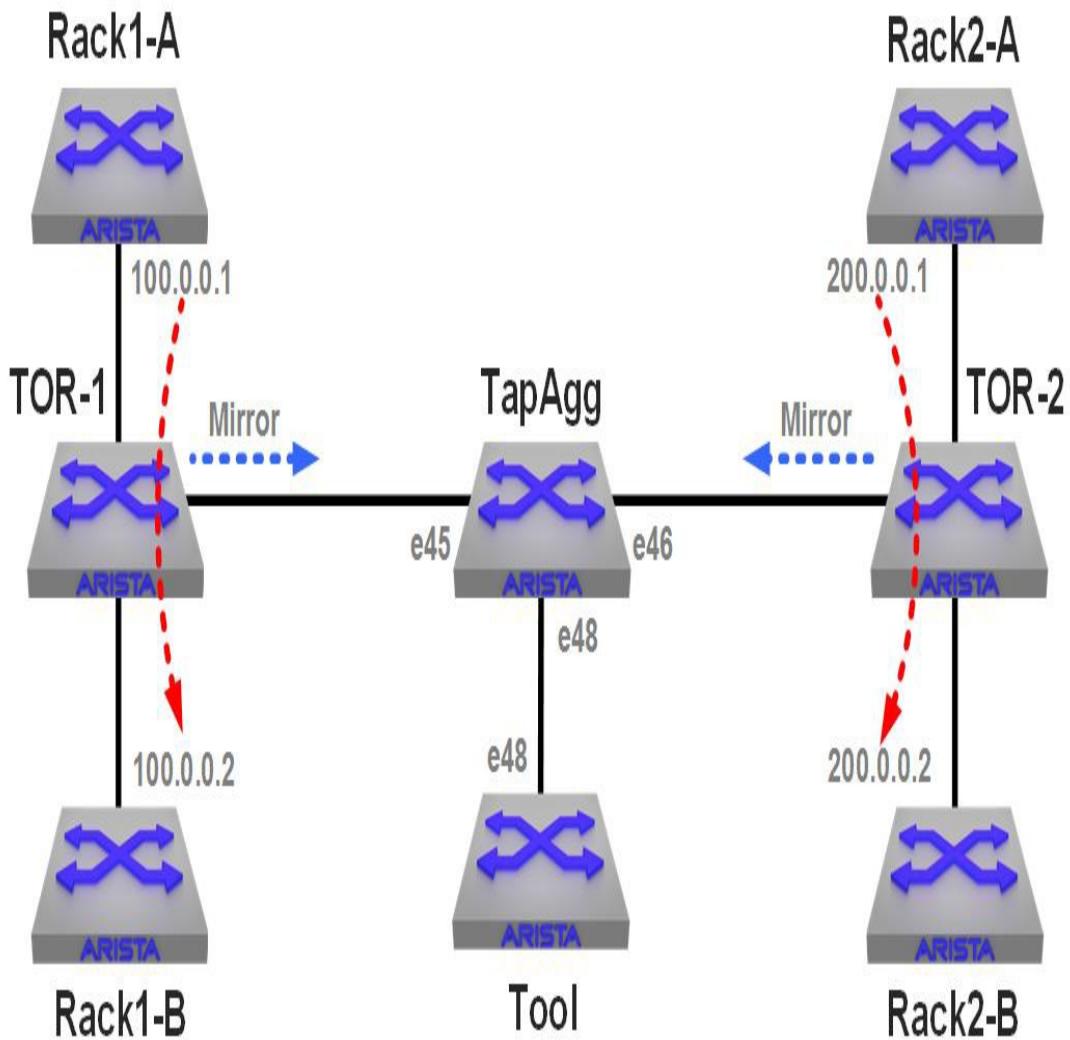


Figure 26-1. A tap aggregation lab

There are two “racks” in this lab: Rack1 and Rack2. Each of the “A” switches have a continuous ping running to the “B” switch in that rack. Each of the Top-of-Rack (ToR) switches have a VLAN connecting them and a mirror session configured with a filter that allows only the ping traffic to be mirrored. Both of the TOR switches have their mirror destinations set to the TapAgg switch in the middle.

The Tool device on the bottom is an Arista switch, and all I'm doing there is a tcpdump of interface e48.

Here's the relevant configuration on the TapAgg switch:

```
tap aggregation
  mode exclusive
!
interface Ethernet45
  description TOR-1
  switchport mode tap
  switchport tap default group Rack1
!
interface Ethernet46
  description TOR-2
  switchport mode tap
  switchport tap default group Rack2
!
interface Ethernet48
  description TapAgg Tool
  switchport mode tool
  switchport tool group set Rack1 Rack2
```

The link to TOR-1 is in the group Rack-1, the link to TOR-2 is in the group Rack-2, and the tool port (e48) is in both groups.

The tool switch has the following monitor configured:

```
monitor session GAD source Ethernet48 rx
monitor session GAD destination Cpu
```

Running the command `tcpdump -c 5 -q -i mirror0` (capture five packets with quiet output on int e48) on the tool switch yields the following:

```
[admin@Tool ~]$ tcpdump -c 5 -q -i mirror0
tcpdump: WARNING: mirror0: no IPv4 address assigned
tcpdump: verbose output suppressed, use -v or -vv for full protocol
decode
```

```
listening on mirror0, link-type EN10MB (Ethernet), capture size
65535 bytes
14:08:28.961608 28:99:3a:be:9f:92 (oui Unknown) > 28:99:3a:be:9d:d6
(oui Unknown), IPv4, length 98: 100.0.0.1 > 100.0.0.2: ICMP echo
request, id 12207, seq 1068, length 64
14:08:29.507166 28:99:3a:be:9e:20 (oui Unknown) > 28:99:3a:be:a0:70
(oui Unknown), IPv4, length 98: 200.0.0.1 > 200.0.0.2: ICMP echo
request, id 4323, seq 702, length 64
14:08:29.961603 28:99:3a:be:9f:92 (oui Unknown) > 28:99:3a:be:9d:d6
(oui Unknown), IPv4, length 98: 100.0.0.1 > 100.0.0.2: ICMP echo
request, id 12207, seq 1069, length 64
14:08:30.507178 28:99:3a:be:9e:20 (oui Unknown) > 28:99:3a:be:a0:70
(oui Unknown), IPv4, length 98: 200.0.0.1 > 200.0.0.2: ICMP echo
request, id 4323, seq 703, length 64
14:08:30.961590 28:99:3a:be:9f:92 (oui Unknown) > 28:99:3a:be:9d:d6
(oui Unknown), IPv4, length 98: 100.0.0.1 > 100.0.0.2: ICMP echo
request, id 12207, seq 1070, length 64
5 packets captured
5 packets received by filter
0 packets dropped by kernel
```

Looking at the destination IP addresses that I've put in bold, you can see that the Tool switch is receiving packets from the continuous ping in Rack1 and from Rack2.

Now, I go into the TapAgg switch and remove the group Rack2 from the tool port:

```
TapAgg(config-if-Et48)#no switchport tool group Rack2
```

Running the same tcpdump on the Tool switch yields the following:

```
[admin@Tool ~]$ tcpdump -c 5 -q -i mirror0
tcpdump: WARNING: mirror0: no IPv4 address assigned
tcpdump: verbose output suppressed, use -v or -vv for full protocol
decode
listening on mirror0, link-type EN10MB (Ethernet), capture size 65535
bytes
14:13:11.958902 28:99:3a:be:9f:92 (oui Unknown) > 28:99:3a:be:9d:d6
(oui Unknown), IPv4, length 98: 100.0.0.1 > 100.0.0.2: ICMP echo
request, id 12207, seq 1351, length 64
```

```
14:13:12.958897 28:99:3a:be:9f:92 (oui Unknown) > 28:99:3a:be:9d:d6
(oui Unknown), IPv4, length 98: 100.0.0.1 > 100.0.0.2: ICMP echo
request, id 12207, seq 1352, length 64
14:13:13.958886 28:99:3a:be:9f:92 (oui Unknown) > 28:99:3a:be:9d:d6
(oui Unknown), IPv4, length 98: 100.0.0.1 > 100.0.0.2: ICMP echo
request, id 12207, seq 1353, length 64
14:13:14.958879 28:99:3a:be:9f:92 (oui Unknown) > 28:99:3a:be:9d:d6
(oui Unknown), IPv4, length 98: 100.0.0.1 > 100.0.0.2: ICMP echo
request, id 12207, seq 1354, length 64
14:13:15.958864 28:99:3a:be:9f:92 (oui Unknown) > 28:99:3a:be:9d:d6
(oui Unknown), IPv4, length 98: 100.0.0.1 > 100.0.0.2: ICMP echo
request, id 12207, seq 1355, length 64
5 packets captured
5 packets received by filter
0 packets dropped by kernel
```

Now, let's remove the Rack1 group and replace it with Rack2:

```
TapAgg(config-if-Et48)#no switchport tool group Rack1
TapAgg(config-if-Et48)#switchport tool group Rack2
```

As you'd expect, now we get only the pings from the other rack:

```
[admin@Tool ~]$ tcpdump -c 5 -q -i mirror0
tcpdump: WARNING: mirror0: no IPv4 address assigned
tcpdump: verbose output suppressed, use -v or -vv for full protocol
decode
listening on mirror0, link-type EN10MB (Ethernet), capture size 65535
bytes
14:16:53.507598 28:99:3a:be:9e:20 (oui Unknown) > 28:99:3a:be:a0:70
(oui Unknown), IPv4, length 98: 200.0.0.1 > 200.0.0.2: ICMP echo
request, id 4323, seq 1206, length 64
14:16:54.507605 28:99:3a:be:9e:20 (oui Unknown) > 28:99:3a:be:a0:70
(oui Unknown), IPv4, length 98: 200.0.0.1 > 200.0.0.2: ICMP echo
request, id 4323, seq 1207, length 64
14:16:55.507599 28:99:3a:be:9e:20 (oui Unknown) > 28:99:3a:be:a0:70
(oui Unknown), IPv4, length 98: 200.0.0.1 > 200.0.0.2: ICMP echo
request, id 4323, seq 1208, length 64
14:16:56.507595 28:99:3a:be:9e:20 (oui Unknown) > 28:99:3a:be:a0:70
(oui Unknown), IPv4, length 98: 200.0.0.1 > 200.0.0.2: ICMP echo
request, id 4323, seq 1209, length 64
14:16:57.507616 28:99:3a:be:9e:20 (oui Unknown) > 28:99:3a:be:a0:70
(oui Unknown), IPv4, length 98: 200.0.0.1 > 200.0.0.2: ICMP echo
request, id 4323, seq 1210, length 64
```

```
5 packets captured
5 packets received by filter
0 packets dropped by kernel
```

This is very useful, but the simple truth is that most people today don't use tap aggregation from the command-line interface (CLI), preferring instead to use the built-in graphical user interface (GUI) or CloudVision (more on that in a moment). I think using it through the CLI is a great thing to know how to do because web access or CloudVision is not always available in the field.

The TapAgg GUI

Using the CLI is a great thing to know, and I honestly just scratched the surface of what's possible. What most people who are using tap aggregation on an Arista switch do is use the TapAgg GUI.

When tap aggregation is enabled, a website is spun up on the switch that you can go to with your favorite web browser (assuming it's reasonably current) that allows you to configure the feature with a nice graphical interface. On our switch, given the last state I left it, that GUI looks like what you see in [Figure 26-2](#).

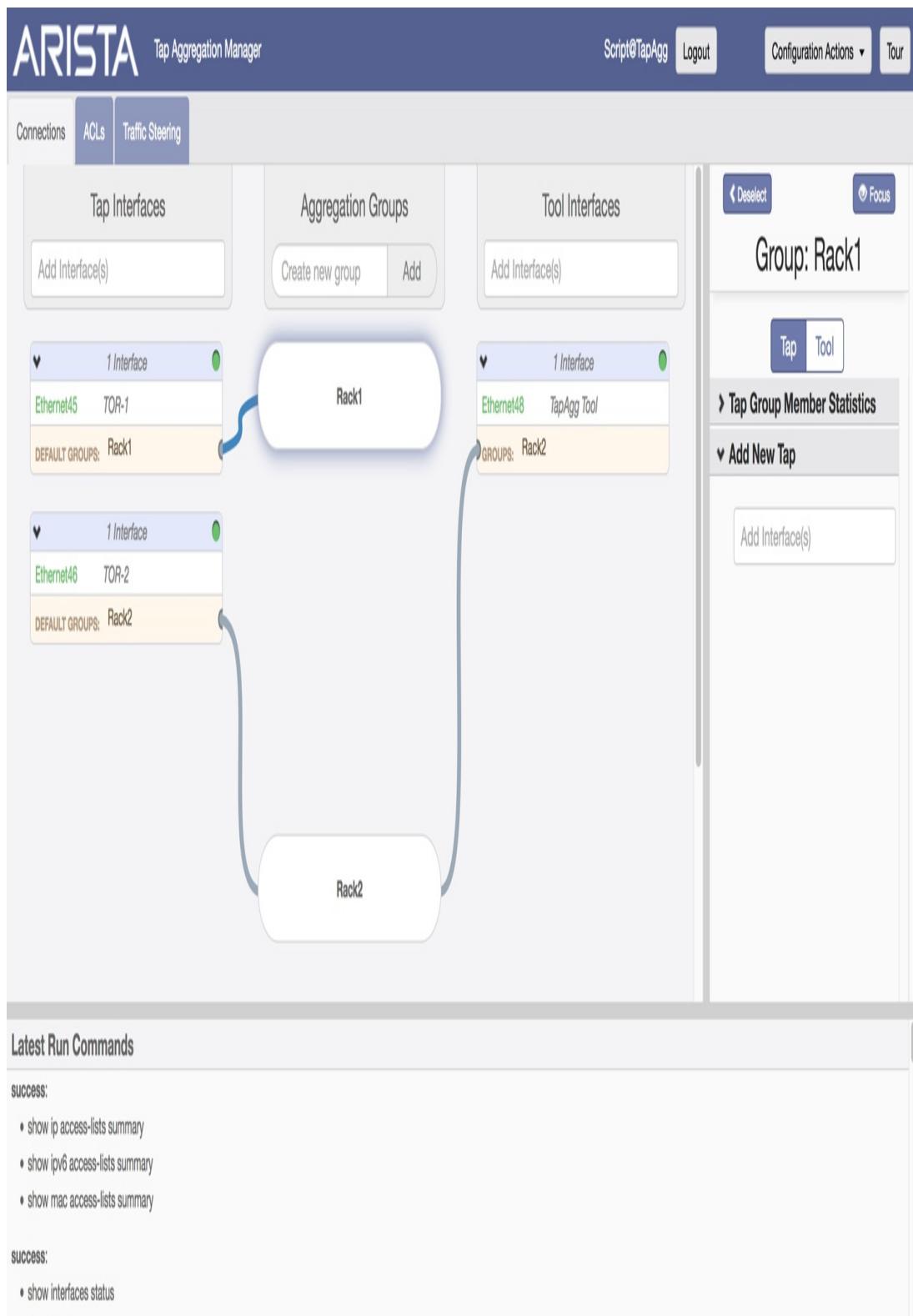


Figure 26-2. The TapAgg GUI

The TapAgg GUI shows a very nice visual representation of what

we've configured and also hints at the fact that we can configure ACLs and Traffic Steering, though I'm not going to go into that level of detail in this book. Of course, the TapAgg GUI is also available through CloudVision if you have tap aggregation enabled on any CloudVision-controlled devices.

Conclusion

Tap aggregation mode is a powerful tool, but remember, on a fixed-configuration switch that it disables normal switching functions for the entire device. On modular switches you can enable individual blades as tap aggregation devices while the rest of the switch remains for your switching pleasure.

Chapter 27. Event Manager

Event Manager is a feature in Arista switches that allows Bash scripts to be executed when certain system events occur. When I first wrote this chapter, I used EOS version 4.9.3, and the triggers were pretty limited. This update is using EOS 4.21.1F with Arista 7280SE-72 or 7280R switches on which some cool new options are included, most of which were added in 4.17.

So, what's the benefit of such a feature? Suppose that you have a system that's been spontaneously rebooting at odd times, and your executive management is too cheap to buy network management software. You could configure an *event handler* to send email (Chapter 22) to you (or the system guys, or whomever you'd like) any time the switch's interface to that server goes down.

Description

Event handlers allow the creation of a *trigger* and an *action*. The trigger in my server example would be the interface on the switch going up or down. The action would be the email being sent. Additionally, we can set a *delay* so that a configured amount of time must pass after the trigger before the action is taken.

NOTE

If you're convinced that you found a mistake because the name of the chapter is Event

Manager but the previous paragraph says event handler, hold your errata, because this is a bit of an Aristicism. You see, the developers made a tool and named it after what it does: it handles events. Marketing then gets a hold of it and named it something that they like better: Event Manager. That's why Event Manager has event handlers.

In the first edition of *Arista Warrior*, when EOS 4.9 was all the rage, only two types of triggers were supported. As of EOS v4.21, there are many more. They are:

on-boot

Triggered when the system boots. Note that this trigger also activates when exiting from event-handler configuration mode.

on-counters

Triggered when certain internal counter thresholds are exceeded.

on-intf

Triggered on certain interface-specific events. Note that this trigger also activates when exiting from event-handler configuration mode.

on-logging

Triggered when regex-matched entries appear in the system log.

on-maintenance

Triggered on maintenance operations.

on-startup-config

Triggered when the startup configuration changes.

vm-tracer

Triggered on VmTracer events.

Let's take a look at each and see how they're configured.

on-boot

The `on-boot` trigger has no other options.

on-counters

The `on-counters` trigger is a very powerful option that's new as of version 4.16 and has significant additions in version 4.17. It can also be switch-dependent because it makes use of Application-Specific Integrated Circuit (ASIC) counters in the switch. This trigger can be a bit difficult to navigate, so let's see if we can make it a bit more palatable.

First, the number of counters that can be used as triggers is enormous. To see them all, use the `sho event-handler trigger counters` command, but be warned: this will create a lot of output. How much output? Piping to the Bash `wc` command with the `-l` option, we can see how many lines of output are produced:

```
Arista#sho event-handler trigger counters | wc -l  
7291
```

Sweet cats! 7,291 lines of output! Each of those lines includes a counter that we can use.

NOTE

The number and type of counters available are greatly dependent on the switch, the ASIC(s) in use on that switch, and possibly the version of EOS in use. For example, doing the same command on a 7280R running 4.21.1F results in 5,596 counters found.

Here's just the first few:

```
Arista#sho event-handler trigger counters | more
```

The list of counters supported by the on-counters event handler

```
Arad0.ActiveQueueCount
Arad0.ActiveQueueCount.delta
Arad0.AvailableSP0ReservedDataBuffers
Arad0.AvailableSP0ReservedDataBuffers.delta
Arad0.AvailableSP0ReservedPacketDescriptors
Arad0.AvailableSP0ReservedPacketDescriptors.delta
Arad0.AvailableSP1ReservedDataBuffers
Arad0.AvailableSP1ReservedDataBuffers.delta
Arad0.AvailableSP1ReservedPacketDescriptors
Arad0.AvailableSP1ReservedPacketDescriptors.delta
Arad0.BadRepliesCtr
Arad0.BadRepliesCtr.delta
Arad0.BdbDynSize
Arad0.BdbDynSize.delta
Arad0.Bfmc1CreditCtr
Arad0.Bfmc1CreditCtr.delta
Arad0.Bfmc2CreditCtr
--More--
```

This output is very different depending on the switch. On a 7280R, I get the following:

```
Student-20#sho event-handler trigger counters | more
```

The list of counters supported by the on-counters event handler

```
Jericho0.Cgm0CgmMulticastDataBuffer
Jericho0.Cgm0CgmMulticastDataBuffer.delta
Jericho0.Cgm0CgmMulticastMaxDataBuffer
Jericho0.Cgm0CgmMulticastMaxDataBuffer.delta
Jericho0.Cgm0CgmMulticastMaxPacketDescriptor
Jericho0.Cgm0CgmMulticastMaxPacketDescriptor.delta
Jericho0.Cgm0CgmMulticastPacketDescriptorDropCnt
Jericho0.Cgm0CgmMulticastPacketDescriptorDropCnt.delta
Jericho0.Cgm0CgmMulticastReplicatedPacketDescriptor
Jericho0.Cgm0CgmMulticastReplicatedPacketDescriptor.delta
Jericho0.Cgm0CgmMulticastReplicationDropQueuePortCnt
Jericho0.Cgm0CgmMulticastReplicationDropQueuePortCnt.delta
Jericho0.Cgm0CgmMulticastSp0DataBuffer
Jericho0.Cgm0CgmMulticastSp0DataBuffer.delta
--More--
```

There is a dizzying array of options, so how do we know which one to pick? Trial-and-error is my favorite blunt instrument, and back on my 7280SE, using it I discovered this:

```
Arista#sho event-handler trigger counters | grep idle
tcollector.proc.stat.cpu.type=idle
tcollector.proc.stat.cpu.type=idle.delta
tcollector.proc.stat.cpu.percpu.type=idle,cpu=0
tcollector.proc.stat.cpu.percpu.type=idle,cpu=0.delta
tcollector.proc.stat.cpu.percpu.type=idle,cpu=1
tcollector.proc.stat.cpu.percpu.type=idle,cpu=1.delta
tcollector.proc.stat.cpu.percpu.type=idle,cpu=2
tcollector.proc.stat.cpu.percpu.type=idle,cpu=2.delta
tcollector.proc.stat.cpu.percpu.type=idle,cpu=3
tcollector.proc.stat.cpu.percpu.type=idle,cpu=3.delta
```

I'll show you how to use these counters in the configuration section later in this chapter.

on-intf

The **on-intf** trigger allows the following trigger types:

ip

Triggered on changes to the interface's IP address assignment.

ip6

Triggered on changes to the interface's IPV6 address assignment.

operstatus

Changes to the interface's operational status (up, down, etc.).

If both the **ip** and **operstatus** triggers are specified, the trigger will be activated when either of them occurs. In other words, specifying them both equals the pseudo-code **ip OR operstatus**, and not **ip AND operstatus**.

To add even more power to this feature, the following variables are passed to Bash when an `on-intf`-triggered action is performed:

`$INTF`

The interface name of the interface specified in the trigger.

`$OPERSTATE`

The current (after the triggered event) operation status of the interface specified in the trigger.

`$IP-PRIMARY`

The current (after the triggered event) primary IP address of the interface specified in the trigger. Note that secondary IP addresses are not included.

`$IP-SECONDARY`

The current (after the triggered event) secondary IP addresses of the interface specified in the trigger. This data is returned as an array.

`$IP6-PRIMARY`

The current (after the triggered event) primary IPv6 address of the interface specified in the trigger. Note that secondary IP addresses are not included.

`$IP6-SECONDARY`

The current (after the triggered event) secondary IPv6 addresses of the interface specified in the trigger. This data is returned as an array.

on-logging

The `on-logging` trigger allows matching the system log against a regular expression (regex). You can set the regex through the `regex` subcommand, and the polling interval with the `poll` subcommand.

on-maintenance

The `on-maintenance` trigger allows you to perform an action when the switch either enters or exits maintenance mode and can be further refined to maintenance mode in Border Gateway Protocol (BGP), on an interface, or for a unit.

on-startup-config

The `on-startup-config` trigger is used to perform an action when the `write memory` (or `copy run start`) command is used. Note that it does not trigger if you manually edit the `startup-config`.

Actions are Bash commands of any type. If you'd like your action to perform more than one command, write a script in Bash and then reference the script in the action. Many people look at this at first and think that it's a severe limitation, but it's absolutely not. Consider this Bash script:

```
#!/bin/bash
# Shuts down an interface input from CLI

/usr/bin/Cli -c "
enable
configure
interface $1
shut "
```

This is a Bash script using `Cli` (which is now the same as `FastCli`, remember) that issues a series of command-line interface (CLI) commands. The `$1` is a token that means we can pass a value to the script. Here's how it works:

```
Arista(config-if-Et1)#sho int status | grep "Et1 "
```

```
Et1          connected  in Po1    a-full a-1G  1000BASE-T
Arista(config-if-Et1)#bash /mnt/flash/ShutInt.sh e1
Arista(config-if-Et1)#sho int status | grep "Et1 "
Et1          disabled   in Po1    auto    auto  1000BASE-T
```

We can absolutely use something like this as an action in event handler, though the variable `$1` would likely be replaced with `$INTF`.

Configuring Event Handlers

Configuring event handlers is a pretty simple affair. The most difficult part would be writing any scripts that you'd like to call and, if you're using the `on-counter` trigger, determining which counter you'd like to use. Let's take a look and see some real examples.

To configure an event handler, enter the event-handler configuration mode by using the `event-handler event-name` command. The event name is anything you'd like it to be, but I urge you to make the name obvious and related to the event trigger in some way. Here, I configure a simple `event-handler` called `Int-E1-UpDown`:

```
Arista(config)#event-handler Int-E1-UpDown
Arista(config-handler-Int-e10-updown)#
```

This drops me into event-handler configuration mode. From there, I can do a bunch of stuff (I love the word *stuff*, don't you?), as evidenced by the following:

```
Arista(config-handler-Int-E1-UpDown)#?
action      Define event-handler action
asynchronous Set the action to be non-blocking
delay       Configure event-handler delay
threshold   Threshold time window where a number of events should
happen
```

timeout	Set the expected time the action should finish in
trigger	Configure event trigger condition
<hr/>	
comment	Up to 240 characters, comment for this mode
default	Set a command to its defaults
exit	Exit from Event handler configuration mode
help	Description of the interactive help system
no	Negate a command or set its defaults
show	Show running system information
!!	Append to comment

Adding comments to parts of the configuration is a wonderful Arista feature that I encourage you to make use of. Let's add a comment to our event by entering the command **comment** alone on a line. This drops you into comment mode, in which you can type a whole bunch of stuff (yeah! more stuff!) across multiple lines. Enter EOF alone on a line to exit this mode:

```
Arista(config-handler-Int-E1-UpDown)#comment
Enter TEXT message. Type 'EOF' on its own line to end.
Trigger for Int-e10 status change
Added November 2018 by GAD
EOF
Arista(config-handler-Int-E1-UpDown)#
```

Now let's define the trigger. I'd like my action to take place any time the interface's status changes, so I use **trigger on-intf *interface-name* operstatus**:

```
Arista(config-handler-Int-E1-UpDown)#trigger on-intf e1 operstatus
```

With my trigger set, now I need to configure the action to be performed on the trigger being, well, triggered. This is done with the **action bash *bash-command*** command. The command I'm going to use is **email -s "Int \$INTF is now \$OPERSTATE"**. This will send an email with the subject line of *Int Ethernet10 is now linkdown* when the

interface goes down, and *Int Ethernet10* is now *linkup* when it comes up:

```
Arista(config-ha[...]-UpDown)#action bash email -s "Int $INTF is now $OPERSTATE"
```

WARNING

For this example to work, you must configure email. See [Chapter 22](#) for instructions on how to accomplish this. Kudos if you've read that chapter and can see what's wrong already.

Just for fun, I include a delay of five seconds:

```
Arista(config-handler-Int-E1-UpDown)#delay 5
```

While still in event-handler configuration mode, the event is not yet active. To enable it, exit the mode:

```
Arista(config-handler-Int-e10-updown)#exit
Arista(config)#
```

My configured event handler looks like this in the *running-config*:

```
event-handler Int-E1-UpDown
  !! Trigger for Int-e10 status change
  !! Added November 2014 by GAD
  trigger on-intf Ethernet1 operstatus
  action bash email -s "Int $INTF is now $OPERSTATE"
  delay 5
```

That's all great, but let's see what happens when the event is actually triggered. After walking up and pulling the cable from interface e10, the following lines appeared in the switch's log:

```
Arista#sho log last 5 min
Nov 26 00:48:14 Arista Cli: %SYS-5-CONFIG_I: Configured from console by
admin
  on vty7 (10.0.0.100
Nov 26 00:48:22 Arista Cli: %SYS-5-CONFIG_E: Enter configuration mode
from
  console by admin on vty7
(10.0.0.100)
Nov 26 00:49:08 Arista EventMgr: %SYS-6-EVENT_TRIGGERED: Event
handler Int-E1-UpDown was activated
Nov 26 00:49:09 Arista EventMgr:
%SYS-5-EVENT_ACTION_FAILED: Eventhandler action Int-E1-UpDown
did not complete
  with exit code 0:Action returned with exit code 2
```

Well, that doesn't look good!

Looking at the configuration, I'm sure many of you saw my earlier mistake, but I did it on purpose (seriously!) to show that incorrect commands will be freely accepted by the event-handler command.

Let's take a look at the event handler with the `show event-handler` command to see where I messed up:

```
Arista#sho event-handler Int-E1-UpDown
Event-handler Int-E1-UpDown
Trigger: on-intf Ethernet1 on operstatus delay 5 seconds
Threshold Time Window: 0 Seconds, Event Count: 1 times
Action: email -s "Int $INTF is now $OPERSTATE"
Action expected to finish in less than 10 seconds
Last Trigger Detection Time: 2 minutes 53 seconds ago
Total Trigger Detections: 1
Last Trigger Activation Time: 2 minutes 53 seconds ago
Total Trigger Activations: 1
Last Action Time: 2 minutes 48 seconds ago
Total Actions: 1
```

This shows some nice information regarding my configured event, including how long ago it was triggered and how many times it's been

triggered. What we're looking for, however, is why it doesn't work. That answer is in the action command string. The email command I referenced requires a destination email address to work, but I didn't give it one. To fix that, I need to go back into event-handler configuration mode and reenter the action command:

```
Arista#conf
Arista(config)#event-handler Int-E1-UpDown
Arista(config-handler-Int-E1-UpDown)#action bash email -s "Int
$INTD is now
$OPERSTATE" gad@gad.net
Arista(config-handler-Int-E1-UpDown)#exit
```

Now, I go plug in my cable into interface e10 and see what happens:

```
Arista#sho log last 1 min
Nov 26 00:53:03 Arista EventMgr: %SYS-6-EVENT_TRIGGERED: Event
handler Int-E1-UpDown was activated
```

Sure enough, five seconds after insertion, I received the following email:

```
Date: Fri, 25 Nov 2016 19:13:29
From: Arista@gad.net
To: gad@gad.net
Subject: Int Ethernet1 is now linkup
```

Nice! But what about those counters I talked about? Let's set an event handler that triggers on CPU utilization. First, we set up the base event handler:

```
Arista(config)#event-handler CPU-Util
Arista(config-handler-CPU-Util)#trigger on-counters
Arista(config-handler-CPU-Util-counters)#
```

Now, we need to choose a counter. Remember the way we found idle

counters? Let's use that information now:

```
Arista#sho event-handler trigger counters | grep idle
tcollector.proc.stat.cpu.type=idle
tcollector.proc.stat.cpu.type=idle.delta
tcollector.proc.stat.cpu.percpu.type=idle,cpu=0
tcollector.proc.stat.cpu.percpu.type=idle,cpu=0.delta
tcollector.proc.stat.cpu.percpu.type=idle,cpu=1
tcollector.proc.stat.cpu.percpu.type=idle,cpu=1.delta
tcollector.proc.stat.cpu.percpu.type=idle,cpu=2
tcollector.proc.stat.cpu.percpu.type=idle,cpu=2.delta
tcollector.proc.stat.cpu.percpu.type=idle,cpu=3
tcollector.proc.stat.cpu.percpu.type=idle,cpu=3.delta
```

We use that first one and make an event handler based on it. We also add some new commands because CPU on these systems can be very spiky, which is perfectly OK.

In this example, we check one of those counters and trigger only if it reports a value of less than 20 twice within 20 seconds while polling every five seconds (which is kind of a useless thing to report, but it's easy to trigger, which is why I used it).

```
Arista(config)#event-handler CPU-Util
Arista(config-handler-CPU-Util)#trigger on-counters
Arista(config-handler-CPU-Util-counters)#poll interval 5
Arista(config-handler-CPU-Util-counters)#condition
  tcollector.proc.stat.cpu.type=idle < 20
Arista(config-handler-CPU-Util-counters)#action bash wall "Idle <
  20%!"
Arista(config-handler-CPU-Util-counters)#threshold 20 count 2
Arista(config-handler-CPU-Util)#exit
```

Here's the result that popped up on my session after a short while:

```
Arista(config)#
Broadcast message from root@Arista (Tue Jan 15 20:11:44 2019):
Idle < 20%!
```

```
Arista(config)#
```

That's kind of nonintuitive, so here's another way to do it:

```
Arista(config)#event-handler CPU-Util
Arista(config-handler-CPU-Util)#trigger on-counters
Arista(config-handler-CPU-Util-counters)#poll interval 5
Arista(config-handler-CPU-Util-counters)#condition
bashCmd."IDLE=$((100-$(vmstat 1
2 | tail -n 1 | awk '{ print $15 }'))); echo $IDLE" > 80
Arista(config-handler-CPU-Util)#action bash wall "CPU > 80%"
Arista(config-handler-CPU-Util)#exit
Arista(config)#
Broadcast message from root@Arista (Sat Nov 26 02:57:34 2016):
CPU > 80%
Arista(config)#
```

Note that the action is triggered as soon as I typed exit, which unfortunately seems to be the way this feature is coded.

OK, so I'm not sure that's exactly intuitive, either, but it works, so now you have two options that work if you're inclined to build a CPU threshold alarm using event handler.

Now let's try to kick the CPU up a bit with my own utility called *corecrusher*. Note that this is not an Arista utility, and it is definitely not something you should do on a production switch.

NOTE

You can download corecrusher from [my GitHub page](#) along with a bunch of other interesting (and potentially dangerous) Arista-related stuff.

The command will *crush* a number of CPU cores at 100% for the number of seconds listed.

In this example, I consume three CPU cores at 100% for 20 seconds. Note that I've configured event handler to poll the CPU percentage every five seconds. Let's see what happens.

```
Arista#bash /home/admin/corecrusher 3 20
To stop corecrusher before the configured time, use the following
command:
corecrusher stop
Crushing 3 core(s) for 20 second(s)
Arista#
Broadcast message from root@Arista (Sat Nov 26 03:03:25
2016):
CPU > 80%
Removing stat file...
Killing corecrushers...
Killed corecrusher(29523) with signal 15
Killed corecrusher(29524) with signal 15
Killed corecrusher(29525) with signal 15
Killed corecrusher(29526) with signal 15
```

It worked!

Now, this is not the most efficient way of monitoring the CPU (CloudVision for the win!), and honestly given the way that Arista switches (and Linux systems) operate, high CPU is not something that is generally a concern, but I made a thing, and that makes me happy.

All in all, setting up and using an event handler is pretty simple, yet this can be a pretty powerful tool.

How about one more example? I used to work with a guy I nicknamed Cowboy Bob the Network Operator from Hell. One of the reasons he earned such a colorful nickname was that he regularly did things like this:

```
Arista>
Arista>en
Arista#wri
Copy completed successfully.
Arista#conf
Arista(config)#wri
Copy completed successfully.
Arista(config)#int e5
Arista(config-if-Et5)#wri
Copy completed successfully.
Arista(config-if-Et5)#desc I like chaos
Arista(config-if-Et5)#wri
Copy completed successfully.
Arista(config-if-Et5)#
Arista#
```

Looking over his shoulder while he did this one day I had to yell, “Stop it!” after which he gave me the “I don’t want to lose any of my changes” excuse. I had to explain to him that his continued writes after each command meant that I couldn’t easily back out of his changes. People can be difficult to change, so how might I use event handler to manage the problem? By using the `on-startup-config` trigger!

The `on-startup-config` trigger is triggered when the `startup-config` changes, but that's actually just mostly true. You see, if you were to drop into Bash and edit the `startup-config` with your favorite editor, that would not trigger event handler. What will trigger this event handler is any system command that changes the `startup-config` such as `write mem`, `copy run start`, and so-on. Here's an example of this trigger in action:

```
Arista(config)#event -handler CowboyBob
Arista(config-handler-CowboyBob)#trigger on-startup-config
Arista(config-handler-CowboyBob)#action bash
/mnt/flash/CowboyBob.sh
```

Now that we have the EOS commands in place, let's see what's in the

Bash script:

```
[admin@Arista flash]$ more CowboyBob.sh
#!/bin/bash

NEWNAME="startup-config_$(date +'%Y-%m-%d_%H-%M-%S')"
cp /mnt/flash/startup-config /mnt/flash/$NEWNAME
```

Running this script results in making a copy of the *startup-config* with a timestamp following the filename. Here's an example file output from manually running the script:

```
[admin@Arista flash]$ ./CowboyBob.sh
[[admin@Arista flash]$ ls -al start*
-rwxrwx--- 1 root eosadmin 3162 Jan 15 20:24 startup-config
-rwxrwx--- 1 root eosadmin 3162 Jan 15 20:38 startup-config_2019-01-15_20-38-16
```

Now let's see what happens when I follow CowboyBob's example with the event handler enabled!

```
Arista>
Arista>en
Arista#wri
Copy completed successfully.
Arista#conf
Arista(config)#wri
int e5Copy completed successfully.
Arista(config)#int e5
Arista(config-if-Et5)#wri
Copy completed successfully.
Arista(config-if-Et5)#desc I like chaos
Arista(config-if-Et5)#wri
Copy completed successfully.
Arista(config-if-Et5)^(z
Arista#bash ls -al /mnt/flash/start*
-rwxrwx--- 1 root eosadmin 3256 Jan 15 20:41 /mnt/flash/start*
-rwxrwx--- 1 root eosadmin 3162 Jan 15 20:38 /mnt/flash/start*_2019-01-15_20-38-16
-rwxrwx--- 1 root eosadmin 3162 Jan 15 20:41 /mnt/flash/start*_2019-01-15_20-41-27
```

```
-rwxrwx--- 1 root eosadmin 3256 Jan 15 20:41 /mnt/flash/...ig_2019-01-15_20-41-50
```

Wait a minute: I issued four **wri** commands but only added two additional files. Why? Because of the nature of this feature and the fact that there is an inherent delay in processing the triggers could not keep up with my cowboy-speed flurry of writes. To remedy this, I'm going to add a **delay 0** to my event handler because there is a built-in delay:

```
Arista(config)#event-handler CowboyBob
Arista(config-handler-CowboyBob)#delay 0
Arista(config-handler-CowboyBob)#exi
Arista(config)#
```

Let's try again:

```
Arista>
Arista>en
Arista#wri
Copy completed successfully.
Arista#conf
Arista(config)#wri
int e5Copy completed successfully.
Arista(config)#int e5
Arista(config-if-Et5)#wri
Copy completed successfully.
Arista(config-if-Et5)#desc I like chaos
Arista(config-if-Et5)#wri
Copy completed successfully.
Arista(config-if-Et5)#
```

Remember, there were four timestamped configuration files before, and we just issued an additional four write commands, so there should be eight files timestamped total now:

```
Arista(config-if-Et5)#bash ls -al /mnt/flash/start*
-rwxrwx--- 1 root eosadmin 3267 Jan 15 20:45 /mnt/flash/startup-config
-rwxrwx--- 1 root eosadmin 3162 Jan 15 20:38 /mnt/flash/...ig_2019-01-15_20-38-16
```

```
-rwxrwx--- 1 root eosadmin 3162 Jan 15 20:41 /mnt/fla[...]ig_2019-01-15_20-41-27
-rwxrwx--- 1 root eosadmin 3256 Jan 15 20:41 /mnt/fla[...]ig_2019-01-15_20-41-50
-rwxrwx--- 1 root eosadmin 3256 Jan 15 20:45 /mnt/fla[...]ig_2019-01-15_20-45-08
-rwxrwx--- 1 root eosadmin 3267 Jan 15 20:45 /mnt/fla[...]ig_2019-01-15_20-45-22
-rwxrwx--- 1 root eosadmin 3267 Jan 15 20:45 /mnt/fla[...]ig_2019-01-15_20-45-31
-rwxrwx--- 1 root eosadmin 3267 Jan 15 20:45 /mnt/fla[...]ig_2019-01-15_20-45-39
-rwxrwx--- 1 root eosadmin 3267 Jan 15 20:45 /mnt/fla[...]ig_2019-01-15_20-45-49
```

Success!

The rapid proliferation of files does raise an important point, which is that such a script will happily fill up the flash drive, especially given the nature of Cowboy Bob and his maniacal configuration writing. How might you deal with that? You could write some logic similar to Linux's **logrotate**, which would save only the last 10 (or 50 or whatever) files. You could save the file somewhere other than flash, but remember that the filesystem on an Arista switch will go away after a reboot. You could mount a Network File System (NFS) drive or you could copy to a remote location using the Secure Copy Protocol (SCP) or use TFTP or a pile of other options, because if there's one thing Arista gives us a lot of, it's options.

Also, I'd probably consider something like Authentication, Authorization, and Accounting (AAA) logging to be a better solution to the Cowboy Bob problem, but my simple script does produce a usable configuration after every iteration where AAA logging does not.

Showing the Event Handler Status

Showing the status of your event handlers is as simple as using the `show event-handler` command:

```
Arista#sho event-handler
Event-handler DropCountersHandler (BUILT-IN)
Trigger: on-counters delay 0 seconds
  Polling Interval: 60 seconds
  Condition: bashCmd."DropCounterMonitor.py" > 0
Threshold Time Window: 0 Seconds, Event Count: 1 times
Action: DropCounterLog.py -l
Action expected to finish in less than 20 seconds
Total Polls: 7230
Last Trigger Detection Time: 5 days 29 minutes ago
Total Trigger Detections: 1
Last Trigger Activation Time: 5 days 29 minutes ago
Total Trigger Activations: 1
Last Action Time: Never
Total Actions: 1

Event-handler CPU-Util
Trigger: on-counters delay 20 seconds
  Polling Interval: 5 seconds
  Condition: bashCmd."IDLE=$((100-$(vmstat 1 2 | tail -n 1 | awk '{ print
$15
}'))); echo $IDLE" > 80
Threshold Time Window: 0 Seconds, Event Count: 1 times
Action: wall "CPU > 80%"
Action expected to finish in less than 10 seconds
Total Polls: 1
Last Trigger Detection Time: 3 seconds ago
Total Trigger Detections: 1
Last Trigger Activation Time: 3 seconds ago
Total Trigger Activations: 1
Last Action Time: Never
Total Actions: 0

Event-handler Int-E1-UpDown
Trigger: None
Threshold Time Window: 0 Seconds, Event Count: 1 times
Action: email -s "Int $INTD is now $OPERSTATE" gad@gad.net
Action expected to finish in less than 10 seconds
Last Trigger Detection Time: Never
Total Trigger Detections: 0
```

```
Last Trigger Activation Time: Never
Total Trigger Activations: 0
Last Action Time: Never
Total Actions: 0

Event-handler CowboyBob
Trigger: on-startup-config delay 0 seconds
Threshold Time Window: 0 Seconds, Event Count: 1 times
Action: /mnt/flash/CowboyBob.sh
Action expected to finish in less than 10 seconds
Last Trigger Detection Time: 9 minutes 18 seconds ago
Total Trigger Detections: 5
Last Trigger Activation Time: 9 minutes 18 seconds ago
Total Trigger Activations: 5
Last Action Time: 9 minutes 18 seconds ago
Total Actions: 5
```

You can also show individual event handlers by specifying an individual name. Let's do that now with the first one from the preceding output because I didn't enter that one:

```
Arista#sho event-handler DropCountersHandler
Event-handler DropCountersHandler (BUILT-IN)
Trigger: on-counters delay 0 seconds
    Polling Interval: 60 seconds
    Condition: bashCmd."DropCounterMonitor.py" > 0
Threshold Time Window: 0 Seconds, Event Count: 1 times
Action: DropCounterLog.py -l
Action expected to finish in less than 20 seconds
Total Polls: 7232
Last Trigger Detection Time: 5 days 31 minutes ago
Total Trigger Detections: 1
Last Trigger Activation Time: 5 days 31 minutes ago
Total Trigger Activations: 1
Last Action Time: Never
Total Actions: 1
```

Apparently, my switch has a built-in event handler, so that's nice. There's really not much else to see and no other useful `show` commands to talk about, but these do give everything you need to know about what's going on..

Conclusion

Event handlers are a great way to automate tasks based on triggered events on your Arista switches. I've actually used this feature to great effect in both my labs and the real world. I encourage you to play around with it and see what it can do for you in your environment.

Chapter 28. Event Monitor

The Event Monitor on an Arista switch is a slick little tool that, according to the documentation, “writes system event records to local files for access by *SQLite* database commands.” Although this is a technically accurate description, allow me to expand on that a bit.

Event Monitor is a process that records certain common events on the switch. As of EOS version 4.17.2F, the events recorded included changes to the Address Resolution Protocol (ARP) table, the Internet Group Management Protocol (IGMP) snooping table, the Link Aggregation Control Protocol (LACP) table, the MAC address table, the mroute table, and the IP routing table. Modern revisions record even more.

OK, I’ll admit that still sounds boring, but let’s dig into this tool and see what it does and how it might be useful.

Using Event Monitor

The home base for using Event Monitor from EOS is the `show event-monitor` command. As of EOS 4.21.1F, there are a pile of options. In the first edition of *Arista Warrior*, there were only four. That book was also written when EOS 4.9 was current. Here are the options in 4.21.1F:

```
Arista#sho event-monitor ?
```

```
all      Monitor all events
arp      Monitor ARP table events
backup   backed up log files
buffer   local buffer settings
igmpsnooping Monitor IGMP snooping table events
lacp     Monitor LACP table events
mac      Monitor MAC table events
mroute   Monitor mroute table events
route    Monitor routing events
<cr>
```

There are a bunch of tables that we can view, and one very cool option named SQLite. The SQLite option lets us send SQLite commands from EOS to the SQLite database, which, as you'll see, is pretty darn useful.

NOTE

SQLite is a software library that, according to the SQLite website, “implements a self-contained, serverless, zero-configuration, transactional SQL database engine.” In other words, it’s a very simple, scaled-down version of SQL that also happens to be in the public domain. For more information on SQLite, see the project web page. Though you might have not heard of SQLite before, if you’ve ever used an Apple Mac running OS X, an iPhone, or an iPad, you’ve used a product that incorporates SQLite. If you don’t like Apple products, look no further than the Firefox browser, the Thunderbird email client, the Google Chrome browser, or the Android phone operating system for other examples of the widespread use of SQLite.

In an early release of EOS 4.21, the default nature of this feature changed and it is no longer enabled by default. This is by design in an effort to allow customers to decide whether the process should be running instead of consuming resources by default. On older revisions of code, you could just look at the ARP table by using the **show event-monitor arp** command, but if you do that now, you’ll get an error:

```
Arista(config)#sho event-monitor arp
% event-monitor not running
% Database does not exist. Run 'event-monitor sync'
```

Unfortunately, the error message on this revision (the latest as of this writing) is incorrect, and running that command will not enable the database. This is a bug that's been reported, so it will likely be fixed soon, but if you see this error, this is why:

```
Student-19(config)#event-monitor sync
% event-monitor not running
```

To enable the `event-monitor` command, issue the `event-monitor` configuration command by itself:

```
Arista(config)#event-monitor
```

After you do this, the rest of the `event-monitor` commands will function:

```
Arista(config)#sho event-monitor arp
2019-01-02
21:59:31.978200|10.0.0.100|Management1|0c:c4:7a:a8:87:ad|0|added|0
2019-01-02
22:04:01.072546|10.0.0.103|Management1|28:99:3a:be:9b:85|0|added|1
2019-01-03
01:36:13.381413|10.0.0.95|Management1|00:1c:73:28:2f:6c|0|added|2
```

A Note About Versions

In EOS 4.16 the internal behavior of this tool changed a bit, which made using SQL commands via the `sqlite` and `interact` commands problematic, and by problematic, I mean they no longer work. To show what I mean, on EOS 4.15 I could issue the following command to get information about the fields within the ARP table:

```
Arista#sho event-monitor sqlite .schema arp
CREATE TABLE arp( time text, prefix text, intf text, ethAddr text,
static integer, delta text,counter integer UNIQUE );
```

Issuing this command post 4.16 yields the following output:

```
Arista(config)#sho event-monitor sqlite .schema arp
% Cannot read from EventMon DB table
```

In fact, issuing any sort of more complicated `sqlite` command yields a similar output where it used to work on earlier code. If you really want to get that sort of information from Event Monitor, you can use Bash and interact using SQLite3 directly:

```
Arista#bash sqlite3  /var/log/eventMon.db '.schema arp'
CREATE TABLE arp (time text,prefix text,intf text,ethAddr text,static
integer, delta text,counter integer UNIQUE);Arista#
```

Although that may seem unfortunate, especially if you're used to using SQL commands to view your data, the good news is that (with the exception of the `.schema` command) there are better ways to get the information using command-line interface (CLI) commands that don't require knowledge of SQL. It's technically not as flexible, but in my experience, network engineers tend to favor using CLI commands anyway, so this isn't really something to get riled up about.

With that out of the way, let's take a look at each one of these tables.

ARP

Normally, ARP changes are not logged on a switch. Even if they were, scrolling through pages of log entries is not my idea of fun, so the idea of storing these events in a database is a bit appealing to me. It's been a

long time since I was a database guy, so let's see how rusty I am. The way to retrieve the ARP events from the database is by using the `show event-monitor arp` command.

NOTE

Many of the code excerpts in this chapter are wrapped in unnatural-looking ways to fit within the confines of the printed page. Because much of the output from these commands is actually Unix output piped through the EOS CLI, the format and width might not look the way you would expect from a traditional switch operating system.

```
Arista#sho event-monitor arp
2019-01-02
22:04:45.273661|10.0.0.100|Management1|0c:c4:7a:a8:87:ad|0|added|0
2019-01-03
01:31:47.828272|10.0.0.95|Management1|00:1c:73:28:2f:6c|0|added|1
2019-01-09
20:24:33.967351|10.10.18.1|Ethernet31|28:99:3a:be:9c:f8|0|added|2
2019-01-09
20:24:36.011052|10.10.18.5|Ethernet32|28:99:3a:be:9d:42|0|added|3
2019-01-09 22:15:58.869723|10.10.18.5|Ethernet32|||removed|4
2019-01-13 18:53:53.460543|10.10.18.1|Ethernet31|||removed|5
```

My main problem with this tool is that the output is, shall we say, ugly. Well, it's ugly from my “all user interfaces should be elegant and beautiful” point of view. From my “it's SQL, and that's what SQL output looks like” point of view, it's beautiful. I know, I'm a complicated person.

Don't get me wrong, this is very useful information, but I have no idea what some of the values are. For example, I see some fields with zeros in them and some with no values, and I have no idea what that last field in every line represents.

To help figure out what this stuff is, I'll take advantage of a nifty SQLite command that shows me the command used to create the table. This command is called `.schema`, and we can access it by using the `bash sqlite3` command. The database resides in `var/log` with the fairly logical name of `eventMon.db`. Each of the supported Event Monitor types are found in similarly named tables within this database. Here, I'm getting the schema for the `arp` table:

```
Arista#bash sqlite3 /var/log/eventMon.db ".schema arp"
CREATE TABLE arp (time text,prefix text,intf text,ethAddr text,static
integer,
    delta text,counter integer UNIQUE);
```

This output tells me that there was a table created named `arp` that contains the following fields, listed in order. I've also added what they mean for any nonprogrammer types out there:

Time (text string)

The time in which this log entry was added.

Prefix (text string)

The IP address related to the ARP entry.

Interface (text string)

What interface the ARP event occurred on.

Ethernet Address (text string)

The MAC address tied to the IP address (prefix).

Static (integer)

Value is 0 if ARP entry was dynamically learned, and 1 if statically assigned.

Delta (text string)

Typical entries are added and removed.

Counter (unique integer)

Every time an entry is made, it is assigned a counter value as a unique identifier for this record.

Let's see this table in action. First, here is the list of existing ARP entries from one of my switches (which happens to be the one I used for the LANZ lab):

```
Arista-2#sho arp
Address          Age (min)  Hardware Addr  Interface
88.0.0.1          N/A        2899.3abe.a026  Ethernet48
10.0.0.95         N/A        001c.7328.2f6c  Management1
10.0.0.100        N/A        0cc4.7aa8.87ad  Management1
```

For this exercise, I add a static ARP entry to my switch by mapping the IP address 192.168.1.2 to the MAC address learned for 192.168.1.1:

```
Arista-2#conf
Arista-2(config)#arp 10.0.0.222 0014.6aa2.d438 arpa
```

Let's see what the Event Monitor database shows now that I've added the static ARP:

```
Arista-2(config)#sho event-monitor arp
2019-01-16
22:42:42.091148|10.0.0.100|Management1|0c:c4:7a:a8:87:ad|0|added|0
2019-01-16
22:44:19.899883|88.0.0.1|Ethernet48|28:99:3a:be:a0:26|0|added|1
2019-01-17
01:45:24.089620|10.0.0.95|Management1|00:1c:73:28:2f:6c|0|added|2
2019-01-19 20:04:58.469700|88.0.0.1|Ethernet48|||removed|3
2019-01-19
20:05:05.205365|88.0.0.1|Ethernet48|28:99:3a:be:a0:26|0|added|4
2019-01-19
20:06:30.330914|10.0.0.222|Management1|00:14:6a:a2:d4:38|1|ad
```

ded|5

Take a look at the line in bold, and let's apply what we know. At 8:06 p.m. on January 19, 2019, a static ARP was added that mapped the IP address 10.0.0.222 to the MAC address 00:14:6a:a2:d4:38. The ARP entry became active on interface Management1.

Now, let's delete the static ARP entry:

```
Arista-2(config)#no arp 10.0.0.222 0014.6aa2.d438 arpa
```

And here's the output from the `show event-monitor arp` command after the change:

```
Arista-2(config)#sho event-monitor arp
2019-01-16
22:42:42.091148|10.0.0.100|Management1|0c:c4:7a:a8:87:ad|0|added|0
2019-01-16
22:44:19.899883|88.0.0.1|Ethernet48|28:99:3a:be:a0:26|0|added|1
2019-01-17
01:45:24.089620|10.0.0.95|Management1|00:1c:73:28:2f:6c|0|added|2
2019-01-19 20:04:58.469700|88.0.0.1|Ethernet48|||removed|3
2019-01-19
20:05:05.205365|88.0.0.1|Ethernet48|28:99:3a:be:a0:26|0|added|4
2019-01-19
20:06:30.330914|10.0.0.222|Management1|00:14:6a:a2:d4:38|1|added|5
2019-01-19 20:08:09.111484|10.0.0.222|Management1|||removed|6
```

Reading the line in bold, we can see that at 8:08 p.m. on January 19, 2019, the ARP entry for the IP address 10.0.0.222 was removed. The ARP entry was previously active on interface Management1.

Remember that ARP entries are made on the switch only when the switch communicates directly using IP or if the ARP table is manually manipulated. If you have no IP addresses active on your switch, this

table will be empty. For example, if you manage your switches solely through console servers and don't even use the Management interface, you might not need to have IP addresses active on your switch at all. In this case, there would be no ARP activity to record.

Event Monitor includes CLI commands to help you organize this output in various ways without having to resort to grep or arcane Bash commands (which I recommend you learn, anyway). For example, you can change how the output is grouped instead of showing the default order, which is time based. This is done by using the **group-by** keyword, which as of EOS 4.21.1F has the options **interface**, **mac**, and **ip**. If we group by IP address, our output changes to look like this:

```
Arista#sho event-monitor arp group-by ip
2019-01-02
22:04:45.273661|10.0.0.100|Management1|0c:c4:7a:a8:87:ad|0|added|0
2019-01-03
01:31:47.828272|10.0.0.95|Management1|00:1c:73:28:2f:6c|0|added|1
2019-01-13 18:53:53.460543|10.10.18.1|Ethernet31|||removed|5
2019-01-09 22:15:58.869723|10.10.18.5|Ethernet32|||removed|4
```

Why are there fewer lines? Because this is the functional equivalent of the SQLite GROUP BY statement:

```
Arista#bash sqlite3 /var/log/eventMon.db 'select * from arp
GROUP BY prefix;'
2019-01-02
22:04:45.273661|10.0.0.100|Management1|0c:c4:7a:a8:87:ad|0|added|0
2019-01-03
01:31:47.828272|10.0.0.95|Management1|00:1c:73:28:2f:6c|0|added|1
2019-01-13 18:53:53.460543|10.10.18.1|Ethernet31|||removed|5
2019-01-09 22:15:58.869723|10.10.18.5|Ethernet32|||removed|4
```

You can also show only specific matches using various match statements from CLI:

```
Arista#sho event-monitor arp match-interface e31
2019-01-09
20:24:33.967351|10.10.18.1|Ethernet31|28:99:3a:be:9c:f8|0|added|2
2019-01-13 18:53:53.460543|10.10.18.1|Ethernet31|||removed|5
```

Using CLI like this is actually a very cool option because it allows abbreviation, which SQLite does not. Notice that I used e31 as the interface name. That doesn't work using SQLite directly:

```
Arista#bash sqlite3 /var/log/eventMon.db "select * from arp
where intf='e31';"
Arista#
```

To make that work we need the exact name as referenced in the table:

```
Arista#bash sqlite3 /var/log/eventMon.db "select * from arp
where
  intf='Ethernet31';"
2019-01-09
20:24:33.967351|10.10.18.1|Ethernet31|28:99:3a:be:9c:f8|0|added|2
2019-01-13 18:53:53.460543|10.10.18.1|Ethernet31|||removed|5
```

Hell, just looking at that SQL statement is probably enough for the typical network engineer to say, “yeah, I’m not doing that,” so having the CLI commands that wrap that up into network engineer-friendly syntax is wonderful. You can also match on *IP address*, *MAC address*, and *time*.

MAC

MAC changes are logged any time a MAC address is learned, added, or deleted. This can happen a lot on a busy switch, but it might not happen much at all on a smaller, stable network. This can be a pretty useful tool, so let’s look into how you can use it by taking a look at the table the same way we did for the ARP table. First, let’s look at the

schema via the `.schema mac` SQLite command:

```
Arista-2#bash sqlite3 /var/log/eventMon.db ".schema mac"
CREATE TABLE mac (time text,fid integer,ethAddr text,intf text,type
text,delta
text,counter integer UNIQUE);
```

Here's the breakdown of the fields:

Time (text string)

The time at which this log entry was added.

FID (integer)

FID stands for Filter ID. Switches keep an internal database of VLANs, and Arista switches assign internal VLANs to certain ports. This number will likely be unimportant to you because the interface on which the MAC address was learned is also present.

Ethernet Address (MAC) (text string)

The MAC address related to this event.

Interface (text string)

The interface on which this MAC event occurred.

Type (text string)

How did the entry change? A typical value would be `learnedDynamicMac`. When a MAC address is removed from the table, this field is null. If you statically add a MAC address by using the `mac address-table static` command, this field will read `configuredStaticMac`.

Delta (text string)

How did the entry change? Typical entries are `added` and `removed`.

Counter (unique integer)

Every time an entry is made, it is assigned a counter value as a unique identifier for this record.

So, let's look at an example of how I like to use this feature. Generic Bob comes up to you at 4:30 p.m. on a Friday and says, "Hey, the network sucks."

"What is it this time, Bob?" you ask, trying really hard not to roll your eyes, since you read that chapter in *Network Warrior* about how not to be *that guy*.

"My system doesn't work, and it used to. Nothing changed, and I don't feel like thinking, so it has to be the network."

After a heavy sigh, you stop the important work you were doing and focus on Generic Bob's dilemma. "Where are you connected, Bob?"

"How should I know? You're the network guru, and it's your crappy network, so you figure it out."

Repressing the thoughts that are better left to mystery writers and psychopaths, you ask, "Well, do you know your IP address?"

"No, I think it's dynamic or something. Will this take long?"

"Look Bob, I need either the IP address or the MAC address to find your system. Get me either of those, or even better, both, and I'll see what I can find, OK?"

Generic Bob storms off to do his generic thing while you pull up your

résumé. Just as you begin to include all of your Arista switch experience, Bob comes back and thrusts a piece of paper at your face. “Here’s your *Mick* address.”

You smile the false, hopeless smile of the damned, gently take the paper, and log in to your Arista switch. Generic Bob’s coffee-stained paper has 3c:07:54:43:88:d4 scribbled on it in what appears to be red crayon, or maybe lipstick.

With Event Monitor, you have a record of every MAC event for quite some time. Using the `match-mac` keyword, you pull up the `event-monitor mac` table including only events regarding the MAC address you care about:

```
Arista#sho event-monitor mac match-mac 3c:07:54:43:88:d4
2018-11-26
09:18:08|1006|3c:07:54:43:88:d4|Ethernet24|learnedDynamicMac|added|549
2019-01-03
16:19:51|1|3c:07:54:43:88:d4|Ethernet6|learnedDynamicMac|added|553
```

Looking at this data, you can see that Generic Bob’s MAC address was first learned on Ethernet24 on November 26, 2018. Because you know that e24 is an uplink to another switch, you know that Bob’s system was originally on another switch.

But the next line shows that the MAC address was learned today (the first day back from work after the new year), at 4:19 p.m. on the switch’s local interface, e6. That seems odd, so you look at the `running-config` for e6 on this switch:

```
Arista#sho run int e6
interface Ethernet6
```

```
description [ Unused ]  
switchport access vlan 999
```

Because you've configured the switch with a dead VLAN (999) that you put on all unused ports, you're sure that Generic Bob glommed a free interface on a different switch than the one to which he was assigned, likely due to either a massive hangover or a natural tendency for Generic Bob to be a pain in the ass. Or both. Yeah, it was probably both.

With a couple of short commands, you've shown that Generic Bob was lying because someone had clearly moved his system from one switch to another, and that's why it stopped working. You can't blame the guy for thinking the new Arista switches would give him better performance, but plugging into network ports without prior authorization should not be tolerated.

I'll leave it to your imagination as to how you might deal with Generic Bob now that you have the tools to prove him wrong. Just try not to be *that guy*...much. I'd probably just smile and tell him to put his system back into the network port where it belonged, and then make sure his access to the data center was revoked, but not until I was done adding Event Monitor to my résumé.

NOTE

This story is a complete fabrication. I've never met anyone named Bob who wrote a Mick address in red crayon. I did know someone who wrote her address in lipstick once, but that's a story for another book. OK, that's actually a fabrication, too. Now I'm sad.

As a fun aside, I asked the Arista customer engineers if they saw this feature being used in the wild and got a pretty significant response saying that it was. I also got a reply that basically reinforced that there are Generic Bobs out there and that this feature managed to save a network engineer's job after said engineer was blamed for an outage that wasn't his fault. Now *that's* a useful feature, and I bet that engineer has been praising Arista ever since.

Route

If you have your switches configured for IP routing, the `show event-monitor route` command can be a great tool to see historical information regarding route changes. As with the others, let's take a look at the schema and then work up an example to see it in action:

```
Arista#bash sqlite3 /var/log/eventMon.db '.schema route'  
CREATE TABLE route (time text,prefix text,protocol text,metric  
integer,preference  
integer,delta text,counter integer UNIQUE);
```

Here's the breakdown of the fields from the `route` table:

Time (text string)

The time in which this log entry was added.

Prefix (text string)

The IP prefix, including the Classless Internet Domain Routing (CIDR) mask related to the route entry.

Protocol (text string)

I would imagine that this should read the protocol from which the prefix was learned or removed, but I've only ever seen this field

read invalid when a route is added, or null when a route is removed.

Metric (integer)

Again, I would assume this field should contain the metric for the route, but I've never been able to make it display anything other than 0 when adding a route, and null when removing a route.

Preference (integer)

This field should include the administrative distance for the route, but in my testing, this field reports a 1 regardless of route type learned.

Delta (text string)

How did the entry change? Typical entries are `added` and `removed`.

Counter (unique integer)

Every time an entry is made, it is assigned a counter value as a unique identifier for this record.

Here is a sample output from one of my switches:

```
Arista#sho event-monitor route
2019-01-22 18:48:51.911482|127.0.0.0/8|martian|0|1|added|0
2019-01-22 18:48:51.911524|127.0.0.1/32|martian|0|1|added|1
2019-01-22 18:48:51.911565|0.0.0.0/8|martian|0|1|added|2
2019-01-22 18:48:51.931530|10.0.0.255/32|receiveBcast|0|0|added|3
2019-01-22 18:48:51.931616|10.0.0.0/32|receiveBcast|0|0|added|4
2019-01-22 18:48:51.931685|10.0.0.0/24|connected|1|0|added|5
2019-01-22 18:48:51.931749|10.0.0.14/32|receive|0|0|added|6
```

Wait, is my switch talking to Martians?

In networking parlance, a *Martian Packet* is a packet seen on the public internet using a private address that is thus considered to be “not of this Earth,” so the packets must be sourced from Martians. Usually they’re from hackers or poorly configured Border Gateway Protocol (BGP)

routers, but calling them Martians is much more fun.

To take that a step further, *Bogons* are Martian IP addresses and IP addresses that have not been allocated (0.0.0.0/8 and 169.254.0.0/16 being two examples) and as such should also not be seen on the public internet. It has become best practice to filter Bogons on BGP edge routers because they're usually associated with hacker attacks.

NOTE

My favorite thing about Bogons is that they get their name from being the quantum of *bogosity*, which is the property of being bogus.

The terms Bogon and Martian are essentially interchangeable today, at least when it comes to internet routing tables. From the point of view of our switch IP routing tables, though, things like 0.0.0.0/8 and 127.0.0.0/8 should only be seen locally, whereas 10.0.0.0/8 might appear from elsewhere on the network. The important distinctions are scale and point of view. The switch might not have anything to do with the internet, so the RFC1918 addresses might very well be legitimate, whereas routes to localhost certainly would not be.

Let's begin with a simple example. Here, I add a static route to the switch:

```
Arista(config)#ip route 20.20.20.0/24 10.0.0.1
```

Using the `match-ip ip-address` option, here's what shows up in the Event Monitor:

```
Arista(config)#sho event-monitor route match-ip 20.20.20.0/24
2019-01-22 20:27:09.835624|20.20.20.0/24|staticConfig|0|1|added|7
```

Now I delete that route:

```
Arista(config)#no ip route 20.20.20.0/24 192.168.1.1
```

And an appropriate remove message shows up in the Event Monitor:

```
Arista(config)#sho event-monitor route match-ip 20.20.20.0/24
2019-01-22 20:27:09.835624|20.20.20.0/24|staticConfig|0|1|added|7
2019-01-22 20:29:00.837986|20.20.20.0/24|||removed|8
```

When the switch learns routes through other means, the results are pretty much the same, though the *Protocol* field will of course change to match how the route was learned.

LACP

LACP (802.3ad) is defined as an IEEE standard that, so far as I can tell, cannot be easily read on the internet due to the IEEE wanting us to pay for the specifications to this open standard. That is infuriating, so I'll leave it to you to determine whether there are any number of ways to get around that while I sit here and grumble to myself about the IEEE and its supposedly open standards.

The output of this `event-monitor` is filled with things that might not make much sense if you don't have a copy of the IEEE specification, and to make matters more complicated, the internal Arista documents state, "The primary purpose of logging LACP events is to allow for post-mortem failure analysis of the Receive and Mux State Machines..." Quick, what is the purpose of the mux State Machine in

LACP? No problem; I'll just check out my copy of the IEEE 802.3ad specification...oh, right.

I have included only a small sample output for the **show event-monitor lacp** command because the resulting text is so wide that there was really no hope of formatting it well in a book. Here are a couple of lines to show what I mean:

```
DC4-2(config)#sho event-monitor lacp
2019-01-22
21:16:07.444659|Ethernet33|unknown|rx|rxSmInitialize|12|0|0|0|0|0|
selectedStateUnselected|0|1
2019-01-22
21:16:07.444815|Ethernet33|unknown|rx|rxSmPortDisabled|1|0|0|0|0|0|
selectedStateUnselected|0|2
[--output truncated--]
```

This command is a little different from the other **event-monitor** options in that it also has a verbose option. Note that the output has been significantly altered to fit on the page. I've altered the date format and collapsed the width considerably:

```
DC4-2#sho event-monitor lacp verbose
Time      Port   LAG      State       Reason
-----  -----  -----  -----
21:16:07  Et33      Initialize  Initializing state machine from Sysdb
state
21:16:07  Et33      PortDisabled  Unconditional transition
21:16:07  Et33      LacpDisabled  portEnabled changed to TRUE with
lacpEna=FALSE
21:16:07  Et33 Po1      Expired    LACP has been re-enabled
21:16:07  Et48      Initialize  Initializing state machine from Sysdb
state
21:16:07  Et48      PortDisabled  Unconditional transition
21:16:07  Et48      LacpDisabled  portEnabled changed to TRUE with
lacpEna=FALSE
21:16:07  Et48 Po1000  Expired    LACP has been re-enabled
21:16:07  Et47      Initialize  Initializing state machine from Sysdb
```

```

state
21:16:07 Et47      PortDisabled Unconditional transition
21:16:07 Et47      LacpDisabled portEnabled changed to TRUE with
lacpEna=FALSE
21:16:07 Et47 Po1000 Expired      LACP has been re-enabled
[--output truncated--]

```

Here's what a couple of lines look like wrapped with their original widths and content:

```

DC4-2#sho event-monitor lacp verbose
Time          Port  LAG   State      Reason
-----
-----
2019-01-22 21:16:07.444659  Et33      Initialize  Initializing state
machine
                                         from Sysdb state
2019-01-22 21:16:07.444815  Et33      PortDisabled  Unconditional
transition
2019-01-22 21:16:07.478546  Et33      LacpDisabled  portEnabled changed
to TRUE
                                         with
lacpEnabled=FALSE
2019-01-22 21:16:07.479061  Et33  Po1      Expired      LACP has been re-
enabled
[--output truncated--]

```

As you can see, the verbose option is well named, but it's also significantly more useful, at least in my opinion. Here is the table layout for LACP in Event Monitor:

```

Arista(config)#bash sqlite3 /var/log/eventMon.db '.schema lacp'
CREATE TABLE lacp (time text,intf text,portchannel text,statemachine
text,state
text,reason int16,portenabled int16,lacpenabled
int16,currentwhiletimerexpired
int16,lacpdurx int16,portmoved int16,selected text,partnersync
int16,counter
integer UNIQUE);

```

Some of these entries are a bit esoteric due to them being flags or

states, as specified in the LACP IEEE specification that I don't have.

Time (text string)

The time at which this log entry was added.

Interface (text string)

The interface involved in this entry.

Portchannel (text string)

The port-channel interface involved in this entry.

Statemachine (text string)

The section of the `state-machine` involved in this entry. Examples include `rx` and `mux`.

State (text string)

This is the new state after the state change.

Reason (16-bit integer)

Reason for change. What are the reason numbers? Check with the IEEE.

PortEnabled (16-bit integer)

Is the port enabled? 0 = no, 1 = yes.

LacpEnabled (16-bit integer)

Is LACP enabled? 0 = no, 1 = yes.

CurentWhileTimerExpired (16-bit integer)

0 = no, 1 = yes.

LacpDurx (16-bit integer)

Has LACPDU been received? 0 = no, 1 = yes.

PortMoved (16-bit integer)

Has port moved? 0 = no, 1 = yes.

Selected (text string)

`SelectedStateUnselected`, `selectedStateStandby`, or `selectedStateSelected`.

PartnerSync (16-bit integer)

Partner Sync flag 0 = false, 1 = true.

Counter (unique integer)

A unique counter for this entry.

IGMP Snooping

This table is much simpler than the last couple of examples:

```
Arista#bash sqlite3 /var/log/eventMon.db '.schema
igmpsnooping'
CREATE TABLE igmpsnooping (time text,vlan text,ethAddr text,intf
text,delta
text,counter integer UNIQUE);
```

Here's the breakdown of the fields from the `igmpsnooping` table:

Time (text string)

The time in which this log entry was added.

VLAN (text string)

The VLAN number on which this entry occurs.

Ethernet Address (text string)

The MAC address.

Interface (integer)

The interface.

Delta (text string)

How did the entry change? Typical entries are **added** and **removed**.

Counter (unique integer)

Every time an entry is made, it is assigned a counter value as a unique identifier for this record.

MRoute

If you have multicast running, this table will show the changes to the multicast route (ip mroute) table similar to the way that the regular route table changes are logged.

```
Arista#bash sqlite3 /var/log/eventMon.db '.schema mroute'  
CREATE TABLE mroute (time text,vrf text,sourceIp text,groupIp text,intf  
text,intfType text,delta text,counter integer UNIQUE);  
Arista#
```

Here's the breakdown of the fields from the **route** table:

Time (text string)

The time at which this log entry was added.

Virtual Routing and Forwarding (VRF) instance (text string)

The VRF in which this mroute resides.

SourceIP (text string)

The source IP address for this entry.

GroupIP (text string)

The group IP address for this entry.

Intf (text string)

The name of the interface for this entry.

IntfType (text string)

The type of the interface (iif/oif).

Delta (text string)

How did the entry change? Typical entries are added and removed.

Counter (unique integer)

Every time an entry is made, it is assigned a counter value as a unique identifier for this record.

Configuring Event Monitor

The database for Event Monitor is found in the *var/log/* directory. This directory does not survive a reboot, so all of the Event Monitor entries will be lost every time the switch reboots. If you'd like to keep a (sort of) permanent log of these events, you'll need to give Event Monitor a location to store its logs in the */mnt/flash* directory. You can do this by using the `event-monitor backup path file-path` command:

```
Arista(config)#event-monitor backup path em.log
```

Specifying a filename without a path results in the file being placed in *flash*:

```
Arista(config)#dir  
Directory of flash:/
```

-rwx 638234211	Nov 12 2018	EOS-4.20.1F.swi
-rwx 700978970	Nov 12 2018	EOS-4.21.1F.swi

```

-rwx      51365      Jun  8 20:43 arpem.log.1
-rwx       27      Jun  6 20:43 boot-config
drwx     4096      Jun  6 20:48 debug
-rwx     51304      Jun  8 20:43 igmpsnoopingem.log.1
-rwx     51304      Jun  8 20:43 lacpem.log.1
-rwx     51363      Jun  8 20:43 macem.log.1
drwx     4096      Jun  6 20:48 persist
-rwx     51365      Jun  8 20:43 routeem.log.1
drwx     4096      Jun  3  01:20 schedule
-rwx     2893      Jun  6 20:43 startup-config
-rwx       19      Jun  3  01:13 zerotouch-config

1862512640 bytes total (256790528 bytes free)

```

If you specify what you think is a full Unix path, such as `/home/admin/em.log`, you will not get the expected results, because this will translate to `flash:/home/admin/em.log`:

```

Arista(config)#event-monitor backup path /home/admin/em.log
Arista(config)#dir
Directory of flash:/

-rwx     638234211      Nov 12 2018 EOS-4.20.1F.swi
-rwx     700978970      Nov 12 2018 EOS-4.21.1F.swi
-rwx       27      Jun  6 20:43 boot-config
drwx     4096      Jun  6 20:48 debug
drwx     4096      Jun  8 20:44 home
drwx     4096      Jun  6 20:48 persist
drwx     4096      Jun  3  01:20 schedule
-rwx     2893      Jun  6 20:43 startup-config
-rwx       19      Jun  3  01:13 zerotouch-config

3269361664 bytes total (1220587520 bytes free)

```

You can specify a Unix path. The command provides two options when asked:

```

Arista(config)#event-monitor backup path ?
file:  forever log URL
flash: forever log URL

```

The idea of a forever log sounds oddly romantic, like rescuing a dog from the pound and bringing him to his forever home, but what this *file versus flash* choice is offering is the choice between a Unix path (*file:*) or a CLI path (*flash:*). If you do not specify either, the parser will assume *flash:*. That's why our previous command using */home/admin* created a directory in *flash:*. To actually put the file in */home/admin* (which would be dumb, because */home* will not survive a reboot either), use the following command, incorporating *file:* at the beginning of the path:

```
Arista(config)#event-monitor backup path file:/home/admin/em.log
```

That will put the database file where you want it:

```
Arista#bash ls /home/admin
em.log.0
```

NOTE

The EOS device, *file:*, is a pseudodevice that lets you access the Linux filesystem from within the CLI. You can use it anywhere you would use another device name, such as `copy flash:GAD.txt file:/tmp`.

You might have noticed that my files all have a *.0* appended to them, although I did not specify that. Event Monitor will write 500 events to the file, after which it will create a new file, appended with a *.1*. The logs are circular, so when the last log (default *.200*) is reached, the *.0* file is deleted, and new data will be written there. That's why I wrote earlier that these files were *sort of* permanent. To configure how many log files you would like to retain, use the `event-monitor backup`

`max-size number` command:

```
Arista(config)#event-monitor backup max-size ?  
<1-200> maximum number of stored backup logs
```

I'll specify 10 log files, for no other reason than I like the number 10:

```
Arista(config)#event-monitor backup max-size 10
```

To disable Event Monitor completely, use the `no event-monitor all` configuration command:

```
Arista(config)#no event-monitor all
```

To enable only one of the Event Monitor functions, use the `event-monitor` command, followed by the function (`mac`, `arp`, or `route`) that you'd like to enable. You can also disable single functions by negating these commands:

```
Arista(config)#event-monitor route
```

To enable them all again, use the `all` keyword:

```
Arista(config)#event-monitor all
```

Finally, you can configure a maximum buffer size for Event Monitor. The Arista manuals show the buffer size in Kb (kilobits), which I kind of think is a mistake, but if it's not, 1 kilobit = 128 bytes.

You can configure the buffer size from 6 to 50 units in size. If backup logs are configured, they are written to when the buffer becomes full:

```
Arista(config)#event-monitor buffer max-size ?
```

<6-50> maximum size of the primary event monitor log

For most environments, leaving the Event Monitor as is and enabled probably makes the most sense. I see no reason to disable it, and even though you might use it rarely, it can be an invaluable tool when you do need it.

Conclusion

I think Event Monitor is a terrifically useful feature, but I think it's one of those features that can be difficult to appreciate until you've been through a situation for which its use could be beneficial. I could argue that's true of any feature, but since this feature is so unique, I think it's doubly true. My goal is to plant the idea in your mind that something like this is useful. That way, even if you don't use it right away, if in a year's time you come across a need for Event Monitoring, perhaps you'll take a second and think to yourself, "*Dammit, GAD was right.*" I live for those moments.

Chapter 29. Troubleshooting

There are some pretty useful diagnostic tools on Arista switches, some of which we've already covered, such as `tcpdump`. Sometimes, we need to know more detail about what the switch is doing, and that's where performance monitoring comes into play.

Logs

In Linux, you can follow live updates to a log file by using the `tail -f filename` command. You can do something similar when viewing your system log in EOS with the command-line interface (CLI) command `show logging follow`. Syslog messages are stored in Linux where any Linux person would expect to see them, namely, in `/var/log/`:

```
[admin@Arista ~]$ ls /var/log
EosInit          error.log          messages
NorCalInit        eventMon.db       nginx-access.log
Post              fneserver         nginx-error.log
agents            inotifyrun-local.log  ntpstats
btmp              inotifyrun-secure.log  ppp
cli               inotifyrun-sys.log   qt
cron              kernel.debug       secure
eos               lastlog           spooler
eos-console       libvirt           startup-config-output
eos-console-sync logrot.log        tallylog
eos-monitor       maillog          wtmp
eos-monitor-sync mcelog          yum.log
```

`/var/log/messages` contains all of the syslog messages that you would

see on the console. Note that you need `sudo` to view this file:

```
[admin@Arista ~]$ cd /var/log
[admin@Arista log]$ tail messages
tail: cannot open 'messages' for reading: Permission denied
[admin@Arista log]$ sudo tail messages
Nov 29 02:07:34 Arista Stp: %SPANTREE-6-STABLE_CHANGE: Stp state
is now stable
Nov 29 02:07:56 Arista Cli: %SYS-5-CONFIG_I: Configured from
console by admin on vty3 (10.0.0.100)
Nov 29 02:08:01 Arista CROND[10746]: (root) CMD
(/etc/archivecheck.sh &> /dev/null)
Nov 29 02:08:33 Arista Cli: %SYS-5-CONFIG_E: Enter configuration
mode from console by admin on vty3 (10.0.0.100)
[--output truncated--]
```

The filesystem in EOS resides in memory, so anything outside of `flash:` will not survive a reboot. Additionally, some switches can be shipped with solid-state drives (SSDs), and if you have a switch that contains one, the system automatically archives many of the system log files onto that drive, which is `drive:` in CLI and `/mnt/drive` in Linux:

```
[admin@Arista log]$ cd /mnt/drive
[admin@Arista drive]$ ls
aquota.user
archive
lost+found
var_archive.2016-11-21-03:20:02.dir
var_archive.2016-11-21-06:19:05.dir
var_archive.2016-11-21-14:30:02.dir
var_archive.2016-11-21-19:15:02.dir
var_archive.2016-11-27-02:30:02.dir
var_archive.2016-11-27-22:29:02.dir
var_archive.2016-11-27-22:45:02.dir
```

Performance Monitoring

A great tool on Linux systems is the `top` command. The `top` command produces output that automatically updates every few seconds (the

default is three seconds in Linux). You can call this command from the CLI with the `show process top` command, or from Bash with the command `top`. Here's a sample output from a live 7280SE-72. This is a very healthy switch with nothing unusual going on. Depending on the switch platform you're using, the processes near the top might change:

```
Arista(config)#sho proc top once
top - 02:22:23 up 1:42, 1 user, load average: 0.28, 0.29, 0.29
Tasks: 289 total, 1 running, 288 sleeping, 0 stopped, 0 zombie
%Cpu(s): 7.0 us, 1.2 sy, 0.0 ni, 91.5 id, 0.1 wa, 0.1 hi, 0.0 si,
0.0 st
KiB Mem: 3796192 total, 3640996 used, 155196 free, 225716 buffers
KiB Swap: 0 total, 0 used, 0 free, 1909320 cached

      PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM     TIME+   COMMAND
 3853 root      20   0 985m 264m 118m S  7.9  7.1  5:14.88 SandFap
 3463 root      20   0 708m 97m 20m S  4.0  2.6  0:01.45 Aaa
 3882 root      20   0 698m 114m 37m S  4.0  3.1  0:15.62 SandMact
11881 admin     20   0 25892 12m 9028 R  4.0  0.3  0:00.11 top
  1 root      20   0 7900 3992 2500 S  0.0  0.1  0:02.22 systemd
  2 root      20   0 0 0 0 S  0.0  0.0  0:00.00 kthreadd
  3 root      20   0 0 0 0 S  0.0  0.0  0:00.06 ksoftirqd/0
  5 root      20   0 0 0 0 S  0.0  0.0  0:00.09 kworker/u:0
  6 root      rt  0 0 0 0 S  0.0  0.0  0:00.19 migration/0
  8 root      rt  0 0 0 0 S  0.0  0.0  0:00.19 migration/1
  9 root      20   0 0 0 0 S  0.0  0.0  0:00.00 kworker/1:0
 10 root     20   0 0 0 0 S  0.0  0.0  0:00.06 ksoftirqd/1
 11 root     20   0 0 0 0 S  0.0  0.0  0:00.07 kworker/0:1
 13 root     rt  0 0 0 0 S  0.0  0.0  0:00.52 migration/2
```

Because I'm a maniac with programming skills, I wrote a program called `corecrusher` that consumes 100% CPU resources for any number of CPU cores. Here I ran it for 30 seconds so that it would consume three cores. Notice that the CPU idle is at only 21.9% and you can see that there are three processes called `corecrusher`, each consuming 99.8% of a CPU core.

```
Arista(config)#sho proc top once
```

```

top - 02:32:00 up 1:51, 1 user, load average: 1.41, 0.60, 0.40
Tasks: 293 total, 5 running, 288 sleeping, 0 stopped, 0 zombie
%Cpu(s): 77.4 us, 0.7 sy, 0.0 ni, 21.9 id, 0.0 wa, 0.0
hi, 0.0 si, 0.0 st
KiB Mem: 3796192 total, 3646380 used, 149812 free, 226364 buffers
KiB Swap: 0 total, 0 used, 0 free, 1911628 cached

          PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM     TIME+ COMMAND
12573 admin      20  0  5040  652  332 R  99.8  0.0  0:27.50
corecrusher
12574 admin      20  0  5040  652  332 R  99.8  0.0  0:27.48
corecrusher
12575 admin      20  0  5040  652  332 R  99.8  0.0  0:27.54
corecrusher
  3853 root      20  0  985m 264m 118m S  8.8  7.1  5:40.01 SandFap
  12601 admin      20  0 25872 12m 9140 R  5.9  0.3  0:00.19 top
  3553 root      20  0  689m  99m  24m R  2.9  2.7  1:40.72 Smbus
    1 root      20  0  7900 3992 2500 S  0.0  0.1  0:02.26 systemd
    2 root      20  0      0      0      0 S  0.0  0.0  0:00.00 kthreadd
    3 root      20  0      0      0      0 S  0.0  0.0  0:00.06 ksoftirqd/0
    5 root      20  0      0      0      0 S  0.0  0.0  0:00.09 kworker/u:0
    6 root      rt  0      0      0      0 S  0.0  0.0  0:00.19 migration/0
    8 root      rt  0      0      0      0 S  0.0  0.0  0:00.19 migration/1

```

There are a variety of ways to sort this output, but the default is by CPU utilization. For 99% of what you'd use this tool for, that's fine. The %CPU column is what determines the process's place on the list, but remember that processes on a Linux system can bounce up and down on this list in milliseconds, and you will likely see just that as you watch.

I recommend doing some further reading on the `top` command online, but there are a couple of things to always look at when you use it. To begin, the very first line shows three numbers after the words `load average:`. These numbers will vary from switch to switch, and they're not necessarily an indication of anything unless you know what they are normally. In other words, a 7280R switch in one environment might run with a load average of (for example) 0.9, whereas another

might run at an average of 1.5. The load average is the average number of processes in the run queue over the interval in question (1 min, 5 mins, 15 mins), and there are so many things that contribute to these numbers that they're not terribly useful on a switch, in my opinion. That being said, if you see a number like 25.99, there's probably something bad happening.

The next bold line in the output shows this:

```
%Cpu(s): 77.4 us,  0.7 sy,  0.0 ni, 21.9 id,  0.0 wa,  0.0  
hi,  0.0 si,  0.0 st
```

This is a very important line, though again, it's only for the snapshot of time when the last `top` iteration ran. In other words, it's not historical; it's a snapshot. The numbers from left to right indicate the following values:

77.4%us

The combined CPUs spent 77.4% of their time on user processes (mostly agents on an Arista switch).

0.7%sy

The combined CPUs spent 0.7% of their time on system processes (kernel, etc.).

0.0%ni

The combined CPUs spent 0% of their time on user processes that have been *niced* (this is a Unix thing used for setting priorities that you really don't need to worry about). Unless you've really messed around in Bash, this should probably always be 0, though there are some processes like *khelper* that pop up with a -20 nice from time to time in order to keep them high priority (a higher niceness means

a lower priority).

21.9%*id*

The combined CPUs are spending 21.9% of their time completely idle. The bigger this number is, the happier the switch is. Note that this is not strictly true, because the forwarding of frames is done by the Application-Specific Integrated Circuit (ASIC) so even with 100% CPU utilization, the switch will likely be operating well.

0.0%*wa*

The CPU spent 0% of its time waiting for I/O to complete.

0.0%*hi*

The CPU spent 0% of its time servicing hardware interrupts.

0.0%*si*

The CPU spent 0% of its time servicing software interrupts.

0.0%*st*

The CPU spent 0% of its time stolen by a hypervisor.

On a healthy switch, *idle* should be high, whereas *system* and *user* should be low. On a busier switch, *user* might spike up and down as processes vie for the CPU's attention. Remember, though, that this is just control-plane stuff like Open Shortest Path First (OSPF), Border Gateway Protocol (BGP), and Spanning Tree Protocol (STP). Packets are forwarded by ASIC, and not the CPU. Still, high CPU should be monitored, and you should contact Arista support if your CPU stays unreasonably high.

Tracing Agents (Debugging)

Debugging processes is significantly different in EOS than it is in other

vendors' industry-standard operating system environments. The amount of debug information available in EOS is staggering, but figuring out where to look can be a bit overwhelming if you're used to using other vendors' debug commands. After you get the hang of how it works in EOS, though, you'll be amazed at the power. You'll also quite possibly never do this unless TAC tells you to, but seeing as how there are limited places to see this documented, I figured it would be a good addition to this book.

First, EOS now has a `debug` command, but it's fairly limited in scope, covering only some basic IP and OSPF topics:

```
Arista(config)#debug ip ?
  general  General debugging
  ospf      Ospf protocol
```

In EOS, the concept of debugging is mostly replaced by the concept of *tracing*. Almost any process (EOS agents) can be traced with—you guessed it—the `trace` command. You can't just go typing `trace ospf packets`, though, so step away from the keyboard and keep reading.

Tracing in EOS can produce an overwhelming amount of data, so by default, the output of your trace does not go to the console or to your Secure Shell (SSH) sessions. Because of the large amount of data, output first must be directed to a file. In this section, I show you how to view the output to the screen via a couple of methods, but let's take a look at how to get a trace started first.

WARNING

Be careful when using `trace`. It can consume system resources, though unlike `debug` in

other vendors' operating systems, it does not take priority over other processes, nor does it stop everything else to deliver its output to the console. (Seriously, who thought that was a good idea?)

Using OSPF as an example, it quickly becomes obvious that you don't trace *protocols* in EOS, but rather *agents*. In fact, all agents have hooks in them for the trace system, which is a pretty clever way for developers to gather information from their agents via built-in methods. To learn the name of the agent that you want to trace, issue the `show agent names` command:

```
Arista#sho agent names
Aaa
Acl
AgentMonitor
AradLanz
Arp
ArpInspection
Asu
Avago
Bfd
BugAlert
CapiApp
Cdp
Chl822X

[ -- lots of output removed in the interest of brevity --]

Vm
VmTracer
Vxlan
VxlanController
VxlanSwFwd
Wbem
Xcvr
XcvrSlice
Xmpp
ZeroTouch
```

Now we have a list of agents that we can trace. One of the frustrating things about this process is that it might not always be obvious what you're looking for. For example, in this list, there are no agents named `Ospf`, `Bgp`, or `Rip`, and the agents that are listed don't have very useful descriptions. That output, however, is from EOS 4.15.5M, and things have changed a bit on newer code.

On newer revisions of code (EOS 4.21.1F in this example), there are actually agents for BGP and OSPF:

```
Arista(config)#sho agent name | egrep -i "ospf|bgp"
Bgp
Ospf
Ospf-vrf
Ospf3
Ospf3-vrf
```

Those agents aren't running by default, though.

```
Arista(config)#sho trace Bgp
% Agent 'Bgp' is not running
```

If you're running *FlexRoute* (Chapter 20) on a supported Arista switch, those agents will be running:

```
Arista(config)#service routing protocols model multi-agent
Arista(config)#sho trace Bgp
Global trace settings for agent Bgp
-----
Tracing sent to stderr

date:      enabled
time:      enabled
[--output truncated--]
```

Back to a default switch, if you know a bit about routing, you might

recognize the idea of a *Routing Information Base* (RIB), and there's a process for that, so let's see where that leads us. To find out details about what you can trace within an agent, use the `show trace agent -name` command. Let's do this for the Rib process:

```
Arista(config)#sho trace Rib
Global trace settings for agent Rib
-----
Tracing sent to stderr

date:      enabled
time:      enabled
PID:       disabled
TID:       enabled
facility name: enabled (width 20)
trace level:  enabled
filename:   disabled (width 20)
line number: disabled
function name: disabled (width 20)

Trace facility settings for agent Rib is
-----
AclAggregator      enabled .....
AclApi             enabled .....
AdjacencyHelper   enabled .....
Agent              enabled .....
AgentIdManager    enabled .....
AgentLogging       enabled .....
AgentStageCommonSm enabled .....
AgentStartupModeSm enabled .....
ArcherBackup       enabled .....
ArcherEm           enabled .....
ArrowATPServerMain enabled .....
ArrowArRT0p         enabled .....
ArrowArRt0p         enabled .....
ArrowArTableInterface enabled .....
ArrowArUtils        enabled .....
ArrowClientLib     enabled .....
ArrowDB             enabled .....
ArrowLocalTable    enabled .....
ArrowProxy          enabled .....
ArrowSerDes        enabled .....
ArrowServer         enabled .....
BfdStatAgentLib    enabled .....
```

```

BfdStaticRouteSm    enabled .....
BgpSmashInfo        enabled .....
ConnectionManager   enabled .....
Dash::Local<bool>  enabled .....

[-- Wow, there's a lot of stuff (removed) --]

Rib                  enabled .....
Rib:::Adv             enabled .....
Rib:::Bgp            enabled .....
Rib:::Bgp::Keepalive enabled .....
Rib:::Bgp::Normal   enabled .....
Rib:::Bgp::Notification enabled .....
Rib:::Bgp::Open    enabled .....
Rib:::Bgp::Policy   enabled .....
Rib:::Bgp::Route   enabled .....
Rib:::Bgp::State   enabled .....
Rib:::Bgp::Task    enabled .....
Rib:::Bgp::Timer   enabled .....
Rib:::Bgp::Update  enabled .....

Rib:::DebugMessages  enabled .....
Rib:::DynPolicyRoutes enabled .....
Rib:::Isis            enabled .....
Rib:::Mio             enabled .....
Rib:::Normal          enabled .....
Rib:::Ospf            enabled .....
Rib:::Ospf3           enabled .....
Rib:::Parse           enabled .....
Rib:::Policy          enabled .....
Rib:::Rip             enabled .....
Rib:::Route           enabled .....
Rib:::RouteControl   enabled .....
Rib:::State           enabled .....
Rib:::Task            enabled .....
Rib:::Te              enabled .....
Rib:::Timer           enabled .....
Rib:::Tracing         enabled .....

[-- Even more stuff removed --]

VxlanIntfAggregator enabled .....
Watchdog             enabled .....
XFire                enabled .....
XFireServer          enabled .....

```

We found it! We're so clever.

NOTE

If you are running *FlexRoute*, the **Rib** process will not be running.

Now that we know the name of the agent (**Rib**), we can begin to specify trace commands. The first step should always be to point the output to a file. You can put the file anywhere, but Arista recommends that it be stored on *flash*: or, better yet, *drive*: if your switch has an SSD drive, given that those locations usually have the most space and can survive a reboot. To do so, use the **trace agent-name filename** *filename* command in configuration mode:

```
Arista(config)#trace Rib filename flash:Rib.txt
```

NOTE

Notice how the agent names all begin with capital letters? That's not a requirement on the command line for specifying the agent name. I'm just obsessive. If you specify a filename with a capital letter like I have, that filename will need to be consistent when referencing it later.

Now there are two paths you can take. You can either trace it all (generally a bad idea), or you can restrict what you want to see. Because we're looking to trace BGP, we need to include only the items with BGP in the name, which I've taken the liberty of highlighting in bold. Actually, there are a couple more, so let's try a different way to see them:

```
Arista(config)#sho trace Rib | grep Bgp
```

BgpSmashInfo	enabled
GatedBgpHelper	enabled
Rib::Bgp	enabled
Rib::Bgp::Keepalive	enabled
Rib::Bgp::Normal	enabled
Rib::Bgp::Notification	enabled
Rib::Bgp::Open	enabled
Rib::Bgp::Policy	enabled
Rib::Bgp::Route	enabled
Rib::Bgp::State	enabled
Rib::Bgp::Task	enabled
Rib::Bgp::Timer	enabled
Rib::Bgp::Update	enabled
RibBgpPlugin	enabled
RibGatedBgp	enabled

Note that these `Rib::Bgp` entries appear only if BGP is running.

To include only these entries, use the `trace agent-name enable trace-facility-name levels` command. Let me show you how to figure all that out. First, we know the agent name (`Rib`), so let's begin there:

```
Arista(config)#trace Rib enable ?
WORD  Trace facility name
```

The `Trace facility name` is one or more entries from that long list we got from using the `show trace Rib` command. Look at the leftmost lines in bold from that output. They all begin with `Rib::Bgp::`. To include them all, we can use a wildcard and specify `Rib::Bgp::*`, as shown in the next example.

NOTE

This is one more place where capitalization matters. If you specify `rib::bgp::*` (no caps), it won't work. If you use caps the way you might think you should such as `RIB::BGP::*`, that also will not work. You need to specify exactly what's listed from the output of `show`

```
trace agent-name, including proper case and the right number of colons.
```

That leaves one more item to determine: *levels*.

```
Arista(config)#trace Rib enable Rib::Bgp::* ?
  all      Enable tracing at all levels
  levels   Enable tracing at one or more levels
```

This one is easy, just use **all**. Sure, you can drill down even further if you'd like, and to do so, specify the word **levels** with a question mark (?) to see what the possibilities are:

```
Arista(config)#trace Rib enable Rib::Bgp::* levels ?
  0      Enable tracing at level 0
  1      Enable tracing at level 1
  2      Enable tracing at level 2
  3      Enable tracing at level 3
  4      Enable tracing at level 4
  5      Enable tracing at level 5
  6      Enable tracing at level 6
  7      Enable tracing at level 7
  8      Enable tracing at level 8
  9      Enable tracing at level 9
  coverage  Enable tracing at level coverage
  function   Enable tracing at level function
```

Like I said, just use **all**. (But feel free to mess with each setting to your heart's content. It's not like I can stop you!)

```
Arista(config)#trace Rib enable Rib::Bgp::* all
```

Note that this will alter the output of the `show trace Rib` command to include references to what is being traced:

```
Arista(config)#sho trace Rib | grep Bgp
Trace facility settings for agent Rib is Rib::Bgp::Task/*cf,
```

```

Rib::Bgp::Normal/*cf,Rib::Bgp::Keepalive/*cf,Rib::Bgp::Update/*cf,Rib::Bgp::State/*cf,Rib::Bgp::Timer/*cf,Rib::Bgp/*cf,Rib::Bgp:::
:
Open/*cf,Rib::Bgp::Notification/*cf,Rib::Bgp::Route/*cf,Rib::Bgp::Policy/*cf
BgpSmashInfo      enabled .....
GatedBgpHelper    enabled .....
Rib::Bgp           enabled 0123456789cf
Rib::Bgp::Keepalive enabled 0123456789cf
Rib::Bgp::Normal  enabled 0123456789cf
Rib::Bgp::Notification enabled 0123456789cf
Rib::Bgp::Open    enabled 0123456789cf
Rib::Bgp::Policy  enabled 0123456789cf
Rib::Bgp::Route   enabled 0123456789cf
Rib::Bgp::State   enabled 0123456789cf
Rib::Bgp::Task    enabled 0123456789cf
Rib::Bgp::Timer   enabled 0123456789cf
Rib::Bgp::Update  enabled 0123456789cf
RibBgpPlugin      enabled .....
RibGatedBgp       enabled .....
SrTeBgpPolicyStatus enabled .....

```

The numbers in the right side of the `Rib::Bgp::` entries indicate the logging levels. Because we chose `all` for the logging level, they're all included. Had we chosen only `level 9`, for example, only the `9` would be shown for those entries.

At this point, the trace is active, and provided there is something going on with the agent you've chosen, the file will begin to fill up:

```

Spine-1(config)#dir Rib*
Directory of flash:/Rib*
-rwx          28464      Jan 31 14:15  Rib.txt
3440762880 bytes total (724103168 bytes free)

```

As a quick aside, the `trace rib enable Rib::Bgp::* all` command will be expanded in the *running-config* file to something that looks like this, which is also shown in the output of `show trace Rib:`

```
trace Rib setting Rib::Bgp::Task/*cf,Rib::Bgp::Normal/*cf,  
Rib::Bgp::Keepalive/*cf,Rib::Bgp::Update/*cf,Rib::Bgp::State/*cf,  
Rib::Bgp::Timer/*cf,Rib::Bgp::Open/*cf,Rib::Bgp::Notification/*cf,  
Rib::Bgp::Route/*cf,Rib::Bgp::Policy/*cf
```

Don't worry about that. It's just all of the items you included by using the asterisk wildcard on one line, separated by commas.

A file that's quietly filling with trace output isn't very useful by itself, so let's see what's in it. First, you can monitor the output directly with the **trace monitor agent-name** command. Press Ctrl-C to break out of this output. This is *not* like using the terminal monitor command in that other company's operating system; when you issue the **trace monitor agent-name** command, you cannot use the CLI until you break the output:

```
Arista(config)#trace monitor Rib  
--- Monitoring /mnt/flash/Rib.txt ---  
14:17:08.654507 task_set_option: task BGP_65006.10.10.6.2+179 socket 101  
option  
    TTL(16) value 1  
14:17:08.654538 task_set_option: task BGP_65006.10.10.6.2+179 socket 101  
option  
    VirtualRoutingandForwarding(39): not supported  
14:17:08.654572 task_set_option: task BGP_65006.10.10.6.2+179 socket 101  
option  
    TOS(17) value 192  
14:17:08.654624 task_addr_local: task BGP_65006.10.10.6.2+179 address  
10.10.6.1  
14:17:08.654658 task_set_option: task BGP_65006.10.10.6.2+179 socket 101  
option  
    DontRoute(5) value 1  
14:17:08.654689 BGP NORMAL: bgp_need_peer_resolution: timeout is 0,  
resolution  
    disabled  
14:17:08.654842 task_connect: task BGP_65006.10.10.6.2+179 addr  
10.10.6.2+179:  
    Operation now in progress  
14:17:08.654899 task_timer_uset: timer BGP_65006.10.10.6.2+179_Connect  
<OneShot
```

```
Set Processing> set to offset 2:28 at 14:19:36
14:17:08.654930 bgp_set_connect_timer: 10.10.6.2 (AS 65006) timer started
with
  timeout 148
14:17:08.654986 task_timer_dispatch: returned from
BGP_65006.10.10.6.2+179_
Connect, rescheduled in 2:28 at 14:19:36
14:17:11.654925 task_process_sockets: processing BGP_65006.10.10.6.2+179
fd 101
  event 8220
```

The method I prefer to use is more Unix centric. Using the `tail filename` command shows you the last few lines of a file. For a constantly updating file like this one, the `tail -f filename` command continuously updates until broken (again, with Ctrl-C):

```
Arista(config)#bash tail -f /mnt/flash/Rib.txt
14:19:03.700397 BGP TASK: bgp_recv_msg_job: processed 1 messages.
yield_now no
14:19:03.700426 task_job_delete: delete background job recv msgq job for
task
  BGP.0.0.0.0+179
14:19:03.700518 task_timer_dispatch: calling
BGP_65002.10.10.2.2+60402_HoldTime,
  late by 0.000
14:19:03.700580 task_timer_uset: timer BGP_65002.10.10.2.2+60402_HoldTime
<OneShot Set Processing> set to offset 2:59.999796 at 14:22:03
14:19:03.700629 task_timer_dispatch: returned from
BGP_65002.10.10.2.2+60402_
HoldTime, rescheduled in 2:59.999 at 14:22:03
14:19:07.550984 task_timer_dispatch: calling
BGP.0.0.0.0+179_PolicyCache_Cleanup,
  late by 0.000
14:19:07.551150 task_job_create: create background job DedupPtrCache
Purge for
  task BGP.0.0.0.0+179
14:19:07.551185 task_timer_dispatch: returned from
BGP.0.0.0.0+179_PolicyCache_
Cleanup, timer now inactive
14:19:07.551223 task_job_delete: delete background job DedupPtrCache
Purge for
  task BGP.0.0.0.0+179
14:19:07.551273 task_timer_uset: timer
BGP.0.0.0.0+179_PolicyCache_Cleanup
```

```
<OneShot> set to offset 10 at 14:19:17
```

I like this method because I can pipe using `grep` (which doesn't work with `trace` from the CLI):

```
Arista(config)#trace monitor Rib | grep 65006
% Invalid input
```

Here's what I'm looking for by using Bash commands from `cli`:

```
Arista(config)#bash tail -f /mnt/flash/Rib.txt | grep 65006
14:22:04.655769 task_timer_dispatch: calling
BGP_65006.10.10.6.2+179_Connect,
  late by 0.000
14:22:04.655874 bgp_connect_timeout: BGP_65006.10.10.6.2+179_Connect
14:22:04.655937 bgp_event: peer 10.10.6.2 (AS 65006) old state Active
event
  ConnectRetry new state Connect
14:22:04.655981 bgp_trap_backward: sending trap for 10.10.6.2 (AS 65006)
14:22:04.656247 task_set_socket: task BGP_65006.10.10.6.2+179 socket 101
14:22:04.656296 task_set_option: task BGP_65006.10.10.6.2+179 socket 101
option
  NonBlocking(8) value 1
14:22:04.656339 task_set_option: task BGP_65006.10.10.6.2+179 socket 101
option
  ReUseAddress(3) value 1
14:22:04.656380 task_set_option: task BGP_65006.10.10.6.2+179 socket 101
option
  TTL(16) value 1
14:22:04.656419 task_set_option: task BGP_65006.10.10.6.2+179 socket 101
option
  VirtualRoutingandForwarding(39): not supported
14:22:04.656458 task_set_option: task BGP_65006.10.10.6.2+179 socket 101
option
  TOS(17) value 192
14:22:04.656553 task_addr_local: task BGP_65006.10.10.6.2+179 address
10.10.6.1
14:22:04.656593 task_set_option: task BGP_65006.10.10.6.2+179 socket 101
option
  DontRoute(5) value 1
```

Turn It Off!

Traces stay running even when you're not looking at them, and they will run, forever consuming disk space, unless you shut them off. Here's a directory listing from my switch where I left a few traces running for only an hour!

```
Agent(config)#dir *.txt
Directory of flash:/*.txt

-rwx    122612063      Jan  7 03:50  RIB-Debug.txt
-rwx    152724544      Jan 11 00:22  Rib.txt
-rwx    189626048      Jan 11 00:22  STP.txt

1862512640 bytes total (0 bytes free)
```

That's 45 megabytes of file space chewed up by something that I don't need. Let's not forget that those traces are still running, which means they're wasting system resources.

NOTE

Though I did run out of disk space, that's more likely due to the fact that this is my test lab switch, which has 4 different versions of EOS on the *flash*: drive, each of which is over 500 MB. If you run these Trace files in your home directory (which runs in memory, remember), then you may cause more significant harm by filling up a filesystem in EOS.

Unfortunately, at least as of EOS version 4.21.1F, I know of no way to disable all traces at once like you can with the `undebbug all` command in other operating systems. To disable tracing, use the `no trace agent-name enable * all` command. This works regardless of what other trace facility names you may have chosen, such as `Rib::Bgp::`.

Remember that you might have multiple traces running, too, and you'll need to stop them all individually.

```
Arista(config)#no trace Sysdb enable * all
Arista(config)#no trace Rib enable * all
```

Don't forget that although you've stopped the traces, the files still reside on disk. If you want to save them elsewhere, you can copy them off using a variety of protocols such as FTP, SCP, TFTP, and more.

Here's a list:

```
Arista(config)#copy flash:Rib.txt ?
boot-extensions  Copy to boot extensions configuration
certificate:      Destination file path
drive:            Destination file path
extension:        Destination file path
file:             Destination file path
flash:            Destination file path
ftp:              Destination file path
http:             Destination file path
https:            Destination file path
running-config   Update (merge with) current system configuration
scp:              Destination file path
sftp:             Destination file path
sslkey:           Destination file path
startup-config   Copy to startup configuration
system:           Destination file path
terminal:         Destination file path
tftp:             Destination file path
```

Usually, I've gotten what I've needed from them, so I just delete them, instead:

```
Arista(config)#del Rib.txt
Arista(config)#del Sysdb.txt
```

With the traces disabled, leaving the filename entries cluttering the configuration doesn't make any sense, so as a final step, we can clean

those up too:

```
Arista(config)#no trace Rib filename flash:/Rib.txt
Arista(config)#no trace Stp filename flash:/Sysdb.txt
```

Tracing can seem a bit overwhelming if you're used to other vendors' offerings and the well-known `debug` commands, but I think if you spend some time getting used to this format, you'll quickly discover that it's more powerful than `debug` and certainly more flexible. The fact that it won't lock up the switch pushing logs to the console is a nice bonus, too.

Honestly, though, most people I know only use Trace when TAC tells them to.

CLI Standalone Mode

Here's a neat instance of something not going quite according to plan. In this case, I have a switch with roughly 400,000 static routes (because I'm a menace and that's what I do). I did a `config replace` on it with a base configuration so that I could work on another chapter, which has tied up the CPU removing all of those CLI commands.

When I SSH to the switch during that process, check out what happens:

```
gad@[AW-Lab]:~$ switch-01
Warning: Permanently added 'student-01,10.0.0.1' (ECDSA) to the list of
known hosts.

Last login: Wed Jan 30 21:29:28 2019 from 10.1.0.100

Cannot connect to CLI. Entering standalone shell.
```

Note: Standalone CLI does not share data with other agents, but it can run certain commands such as 'bash' to troubleshoot the system within authentication and authorization constraints.

Standalone>

Whoa! In all the years that I've been working on Arista switches, I'd never seen this. Let's see what we can see!

```
Standalone>en
Standalone#sho run
! Command: show running-config
! device: Student-01 (unknown-hardware, EOS-4.21.1F)
!
! boot system flash:/EOS-4.21.1F.swi
!
prompt Standalone%R%v%P
!
transceiver qsfp default-mode 4x10G
!
spanning-tree mode mstp
!
no aaa root
!
no ip routing
!
end
```

Well that's new. What about interfaces?

```
Standalone#sho int
Standalone#
```

Nothing! There's not even a management interface! The management interface is a kernel interface in Linux, and so it is not controlled by the ASIC, and I can't even see that! Some poking around has found that this is a special `Cli` mode that loads without access to SysDB or any other agents. Here's the relevant help output from the `Cli` command in

Linux:

```
[admin@switch-02 ~]$ Cli --help | grep -A1 "without Sysdb"
-l, --standalone      run in standalone mode (without Sysdb). Implies
                      --disable-guards.(also see --standalone-guards
```

At this point, I mutter, “fascinating” in my best Mr. Spock voice and then try to trigger standalone mode myself. I couldn’t get that to work and quickly gave up because I had other things to do.

What’s the point of this? To show you that if you do see this, not to panic. After about a minute, I was able to SSH to the switch normally, even with my standalone shell still active. I just thought it was an interesting (fascinating, even) curiosity and thought I’d share it.

Bottom line: if you see this, don’t panic. Wait a minute or two and try to SSH again. I bet it will work. If it doesn’t after a reasonable amount of time, something more serious might be afoot, at which point it might be time to call TAC. Or reboot. Maybe panic. It all depends on your environment and caffeine intake.

Arista Support

Arista Support, or TAC (Technical Assistance Center), is a fantastic resource, and my experience with it has been far superior to that other company that has nothing to do with crystallized cottonseed oil. TAC has helped me when I’ve discovered bugs, it has helped me diagnose configuration problems, and on the rare occasion when we had a switch go bad under warranty, TAC shipped us a new one in no time. After you’re registered with Arista Support as a valid customer, opening a TAC case is as simple as calling, or if the issue isn’t critical, you can

use email, as well. And with your valid and registered email address, you can download documentation, EOS revisions, whitepapers, and a host of other useful documents from the Arista.com website.

There is also a fabulous online forum and repository of knowledge at the EOS Central website. At this site, you can post a question to the forum, and you might be surprised to find one of the company founders answering it. There is also a development blog and some tech tips that are well worth a look.

Of course, you can always send me an email, too, but I'm a cranky, old, recalcitrant nerd who doesn't have a lot of free time, so you'd probably do better emailing Arista. That is, of course, unless you want to tell someone how much you like this book, in which case I'm probably the better choice.

Conclusion

The concept of troubleshooting is one that I am asked about quite a bit, and it's a tough topic to write about within a single chapter. The idea of writing about *common things that go wrong* is really what people want, but in my experience, I don't really see *common things that go wrong* aside from my favorite thing to troubleshoot, which is *user error*. My favorite complaint from students and entrenched engineers alike is, "I did everything right, but it still doesn't work."

Chapter 30. eAPI

As you've likely figured out by now, I think most all of the features of EOS are pretty darn cool. The EOS Application Programmable Interface (eAPI), though, has a special place in my heart. Yes, people like me have special places in our hearts for features in network operating systems.

Simply put, eAPI is a means whereby command-line interface (CLI) commands can be executed either via a web interface or, more importantly, through a program or script. Given the power of EOS and the many other means by which the switch can be controlled via custom programs, that might not seem like a big deal, but it is for reasons that should soon become clear. To explain, allow me to begin with a story of pain.

But first, a rant.

GAD's Rant About the Fear of Scripting

I've been a lot of things in my professional career. I went to school to be a programmer back before the job title changed to developer, and I've been a Unix administrator, a generic IT guy, and of course, a network guy. I need to hit you up with some cold-hard truth here, so take it with the understanding that I've seen the industry unfold since the 80s and I have a fair bit of insight into how things are changing.

Ready? Here it is:

WARNING

Learn to code or I will script you out of a job.

The networking world is changing, and I've been preaching for the past six years that there's only 10 years left. Well, do the math as that's no longer enough time. Time for what, you might ask? Time for you to adapt.

I meet a lot of networking professionals in my job, and the number of networking people who don't want to program is staggering. If you are one of those people, learn to code or I will script you out of a job. If you do not adapt, you will be replaced. It's that simple. What's even more surprising to people who can't accept this truth is that they might very well be replaced by automation. That automation can be something as simple as a script. If you can't write those scripts, it will be someone else's script that replaces you. If you can write scripts, it might well be your code that scripts someone else out of their job. Adapt or die. Too harsh? How about adapt and overcome, because if you don't adapt, you will be overcome with the cold-hard truth that someone has replaced you with a script.

In my experience, networking engineers are mostly behind the curve in regard to automation, though this greatly depends on the environment in which they find themselves. When I work with the big tech companies, I see much more acceptance of automation than I do in the

enterprise. This became painfully evident to me at a client site when, after showing the networking engineers all of the automation possibilities built into Arista's EOS, they begged me not to show them to the systems teams. They knew at their core that the systems teams would take to the automation tools instantly because systems admins love automation and have been doing it for decades. Those systems engineers were proficient in a new way of thinking that's commonly called *DevOps*: a portmanteau of development and operations.

If you still don't believe me, consider this: if I walk into a meeting with your CEO and tell her that I can replace 10 of her network engineers with automation, do you think she would pay me one engineer's salary for a month of writing and implementing that automation? You bet she would. Learn to code so you still have value in a DevOps world. Don't think that DevOps is a real threat? Well, guess what, now there's *NetOps* to worry about, too. If your reaction is "my job is too specialized to be replaced by a script," you're in for a rude awakening. If the cloud titans (Google, Amazon, Microsoft, etc.) can automate their environments, your company can, too.

The good news is that, in my opinion, eAPI is a tool that lends itself very well to network engineer adoption because it uses the same CLI commands that we've all spent years learning and understanding. To understand how that all works, I need to take a quick detour into a couple of topics that are commonly called *expect scripting* and *screen scraping*.

Expect Scripting

In another life, I was fervently pursuing the Cisco Certified Internetwork Expert (CCIE) certification in an effort to inflate my own self-worth (not to mention my wallet). To that end, I had spent a fair amount of time and money on a rack of routers and switches in my garage. By “fair amount,” I mean 12 routers and four switches, all in a 19-inch telecom rack that’s properly grounded, has backup power, and is properly bolted to the cement floor in my garage. No, we can’t fit a car in the garage, and yes, my wife hates me.

Spousal friction aside, my rack of gear was great for studying, but since I travel so much, I needed to be able to control it all remotely. Naturally, that meant more equipment, so I added a nice web-enabled power distribution unit and a terminal server. I also had a management switch connecting all of the devices together along with a Linux server where I could store versions of code for all the devices along with copies of various configurations I had created. For those keeping track at home, that’s now 12 routers, four switches, a management switch, a terminal server, and a Linux machine, all totaling 19 rack units (RUs) of space, not including the UPS, firewalls, other servers, and various other devices, all of which work together to make me have to explain why the NSA doesn’t need to be called every time the cable guy comes in for a service call.

The problem with all this gear, especially in a lab/training scenario, is that I often need to load different versions of code, not to mention completely different sets of configurations on the entire rack. For example, if I’m working on the intricacies of OSPF, I might have completely different configurations on every device than I would if I were working on, say, a multicast lab. I needed a way that I could bulk-

load configurations on 16 networking devices with minimal interaction. Sure, I could copy a configuration from *flash*: into the *startup-config* and reboot, but remember, I'd have to do that 16 times. There had to be a better way.

To accomplish what I needed to do, I wrote a special frontend using Perl and the module `Net::Telnet`. Well, to be painfully accurate my good friend Adam Levin wrote the base script for me, and I added literally thousands of lines of code to bend it to my will. I'm not a huge fan of Perl, but Adam is, so I used what he gave me and didn't complain. Much.

On my Linux box, that 20,000-line script would telnet to the terminal-server and then through a complex set of interactions on each device; it would issue commands, wait for the proper response, and thereby control the devices in my lab programmatically. I was thrilled with my solution, but it was clunky. Remember, because my script was essentially controlling each device from the console, I had to write logic that would not only test for each command's output, but also take into consideration that different devices, and even different versions of code, produce potentially different output. My script simulated a user sitting at a terminal.

In its simplest form, this method is called *expect scripting* because your script expects to see output, after which it responds. There is a programming language called *Expect*, but the term is generically used for any similar functionality. A script sends a command, waits for an expected result, and when it gets that result, continues. For example, the task of issuing a `write mem` command would go something like

this:

First, I would need to connect to the proper device through the terminal server. Remember, though, that because this is the console, we would need to consider the state of that port. What if someone had used it and left it logged in? What if they did a `show run` and left it at a `--more--` prompt? Sending a `write mem` to the console while it was sitting at a `--more--` prompt would do nothing other than break out of the `--more--` prompt, thus rendering the rest of the script useless. So, I needed to get the console into a known good state (this script was actually altered for use with EOS, which is why you see reference to an admin user):

```
# $device is the device name on the

$ld->print("$device");
$ld->print('');

# Assumes Cisco TermServ with User/Pass configured
$ld->waitFor("/Username: /");
$ld->print('Script');
$ld->waitFor("/Password: /");
$ld->print('ClearTextGoodness');
$ld->print('');

# send Control-C to get back to command prompt
# Control-C breaks --more-- prompts in CLI
# Also breaks running pings in both CLI and BASH
$ld->print("\cC");

# If in BASH, two "exit"s will get us back to CLI and logged out
# If in CLI, two "exit"s will get us trying to login with user "exit"
# Extra CR breaks us out of login and doesn't hurt if we're not there
$ld->print("exit");
$ld->print("exit");
$ld->print("");

$ld->waitFor("/in:/");
$ld->print("admin");

# If there is a password, this will break. There must be no PW!
```

```
$ld->waitFor("/(>|#)/");
$ld->print('en');
$ld->waitFor("/#/");
```

All of that code is just to get a single device to a known-good state, which is enable mode. Remember that different devices might respond differently, which means that I needed different routines for every class of device in my lab. Now we can send a `write mem` command:

```
$ld->print('write mem');
$ld->waitFor("/#/");
print "<I>Done</I><br>\n";
```

Don't forget that I'd connected through a terminal server, so I'd also needed the following bit of code after I was done, to properly disconnect (assuming a Cisco console server):

```
# Exit from TermServ connection for this device
$ld->print("\c^x");
$ld->print("disco");
$ld->waitFor('/\[confirm\]/');
$ld->print('y');
```

That page of code accomplished one thing: sending the `write mem` command to a device. Not only that, but it worked on a device that used only that exact sequence of commands. What if instead of `Username:` the device responded with `Login:?` Well, in that case, the code `$ld->waitFor("/Username: /");` would wait forever (or until a timeout expired) and the script would hang or error out. I managed to code around that by having a routine for every class of device, and because I bought all of my devices used on eBay, there was a variety of devices. Just think about what happens when you boot an older Cisco device with no configuration. You might get any of the following prompts:

- router#
- localhost#
- switch#
- pix#
- Would you like to enter the initial configuration dialog?

Every one of those possibilities had to be accounted for. After I'd done that, I had a script that would run serially (remember, this was over console ports) across 16 or more networking devices at 9600 baud. When the script ran properly, it would sometimes take 45 minutes to complete! But it did complete, and it was magnificent. Until some idiot left device number nine in the following state:

```
rommon#
```

When that happened, I'd need to go get the switch out of *rommon* and then start the process all over again. It was not efficient, but remember this was the 1990s when things like automation and APIs were a foreign concept in the networking world.

Sadly, that's not even the worst of it.

Screen Scraping

If you think the whole expect scripting thing sounds like a pain, imagine that you finally got all of that to work, and you now wanted to retrieve the version from the `show version` command. Not only would you need to go through all of that expect scripting stuff, but you'd then need to strip out the interesting information from the output of the

command. Getting more advanced results from the output is colloquially called *screen scraping*. Let's see what that means:

Here's the output of `show version` from an Arista switch:

```
Arista(config)#sho ver
Arista DCS-7280SR-48C6-M-F
Hardware version: 21.05
Serial number: SSJ17290598
System MAC address: 2899.3abe.9f92

Software image version: 4.21.1F
Architecture: i386
Internal build version: 4.21.1F-9887494.4211F
Internal build ID: 1497e24b-a79b-48e7-a876-43061e109b92

Uptime: 0 weeks, 0 days, 1 hours and 0 minutes
Total memory: 32458980 kB
Free memory: 30501332 kB
```

Why might I need this in my lab? Suppose that I want to make sure that all the switches were on the same revision of code. I might want to get the currently installed version so that I upgraded only devices that needed to be upgraded (or downgraded as the case may be). I could make that simpler with CLI tools, which is a great first step:

```
Arista(config)#sho ver | grep image
Software image version: 4.21.1F
```

That's certainly better, but how can I get only the information I care about, which in this case is the 4.18.1F portion of the output? In a Python script (I'm changing to Python because Perl makes my teeth itch), I would likely resort to something like this, where `foo` contains the string retrieved through the previous command:

```
print foo.split()[3]
```

This bit of Python takes `foo`, splits it where there's white space, and prints the third field (remember we count from zero). Here's the result:

```
>>> foo = "Software image version: 4.21.1F"
>>> print foo.split()[3]
4.21.1F
```

This has serious limitations, though. What if the next version of EOS changes that line from `Software image version: 4.21.1F` to just `Version: 4.22.0F`? Our line of code that references field 3 of the split output will break and the script won't work. And, yes, that sort of thing happens all the time.

```
>>> foo = "Version: 4.21.1F"
>>> print foo.split()[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

If you think this all sucks, welcome to the club, but before networking equipment had things like Application Programmable Interfaces (APIs), expect scripting and screen scraping were the only way to automate tasks. We're well into the twenty-first century now, and there has to be a better way. Enter Arista's EOS API—eAPI.

eAPI lets you issue CLI commands from a script (or web page, as you'll see) and get the results in a format designed to be read by a program. That format is called JSON, which stands for the *JavaScript Object Notation*. Don't worry, there's no JavaScript involved. Let's take a look.

From the command line, when I issue the `show version` command, I

get this output:

```
Arista#sho ver
Arista DCS-7280SR-48C6-M-F
Hardware version: 21.05
Serial number: SSJ17290598
System MAC address: 2899.3abe.9f92

Software image version: 4.21.1F
Architecture: i386
Internal build version: 4.21.1F-9887494.4211F
Internal build ID: 1497e24b-a79b-48e7-a876-43061e109b92

Uptime: 0 weeks, 0 days, 1 hours and 8 minutes
Total memory: 32458980 kB
Free memory: 30520652 kB
```

But check this out. Since about EOS 4.14, you can view most commands by using a **json** modifier, which takes the output and presents it to you in JSON format:

```
Arista#sho ver | json
{
    "uptime": 4111.17,
    "modelName": "DCS-7280SR-48C6-M-F",
    "internalVersion": "4.21.1F-9887494.4211F",
    "systemMacAddress": "28:99:3a:be:9f:92",
    "serialNumber": "SSJ17290598",
    "memTotal": 32458980,
    "bootupTimestamp": 1546895161.0,
    "memFree": 30520076,
    "version": "4.21.1F",
    "architecture": "i386",
    "isIntlVersion": false,
    "internalBuildId": "1497e24b-a79b-48e7-a876-43061e109b92",
    "hardwareRevision": "21.05"
}
```

JSON generally presents data in a format that works well with most programming languages, and in the case of Python, the data types use the same format, which is nice. Why not use XML? Because XML is

very verbose and generally causes a lot of work to get to the data you're looking for. JSON is much more concise. The biggest plus in my opinion is that JSON presents data in key-value pairs where appropriate. To understand why that's such a good thing, let's take another brief detour, this time into the world of Python data types.

Python Data Type Primer

Before we proceed, let's have a quick lesson on data types using Python as our language of choice. Apologies for shifting from Perl to Python, but all of the examples from now on will be in Python. Why? Because Arista doesn't use Perl (I did at the time because Adam agreed to help me, and Adam likes Perl). Stay with me while I take a step back and explain Python data types, and that should help make clear why eAPI is so darn cool.

Scalar variables are defined and used as you'd expect a simple variable to work. They can hold only one value and are said to be *atomic* because of that fact.

```
foo = 3.14
    foo is type float

foo = 'GAD'
    foo is a string

foo = 6
    foo is an integer
```

Storing single values is certainly useful, but oftentimes we need to store multiple values in a single variable. This type of variable is

generically called an *array* in programming, and there are many types of arrays.

NOTE

A quick note is in order with regard to the Python `print` command. Python 2 uses the `print statement`, whereas Python 3 uses the `print() function`. Because I learned on Python 2, I loathe using `print()`, and, as such, I refuse to do things like `import __future__` to make my code work with Python 3 syntax. I am definitely a loudmouth, opinionated curmudgeon if you haven't figured that out yet. Python 2 `print` rules!

List

In Python, an array of values stored in a single variable in which each variable is referenced by its position in the array is called a *list*. Elements within a list can be of any data type. Lists are defined using square brackets with each element separated by a comma:

```
foo = [ 3.14, 'GAD', 6 ]
```

Lists are referenced using the position of the element starting at zero with the element position in square brackets. To print the string 'GAD' in this example, you would use the following format (remember we count from zero):

```
print foo[1]
```

Tuple

In Python, a read-only (or *immutable*) list is called a *tuple*. Tuples are defined just like lists but with parentheses instead of square brackets:

```
foo = ( 3.14, 'GAD', 6 )
```

Tuples are referenced using the position of the element starting at zero with the element position in square brackets. To print the string ‘GAD’ in this example, you would use the following format:

```
print foo[1]
```

If that seems confusing, you’re not alone in thinking so. The reason the format is a square bracket even though a tuple is defined with parentheses is because you’re not referencing the position in a tuple or a list, but in what Python calls a *sequence*. In Python, lists, tuples, strings, and even dictionaries (stay tuned) are sequences, and all sequences have positions referenced within square brackets. For example, to print just the first (the *zeroth*, remember) character in the string ‘Arista,’ you could do this:

```
>>> foo = 'Arista'  
>>> print foo[0]  
A
```

Dictionary

Lists and tuples are really cool, but they’re all pretty limited because they are ordered and therefore positional (which is often the main benefit). To reference an element in either, you must know its position. In Python, a dictionary is non-ordered, and each element is given a name, which is how you reference that element. The name of the element is its *key*, and the data contained within that element is its *value*. Hence, a dictionary contains key–value pairs. A dictionary is defined using curly braces and the elements are still separated by commas, but each element now has a key and value to worry about. The key comes first and is followed by a colon and then the value associated with that key. The key must be unique within the dictionary. Let’s take a look:

```
foo = { 'pi': 3.14, 'name': 'GAD', 'age': 6 }
```

Python allows all sorts of flexibility when it comes to white space, so I will often write that same line of code like this, which I find far easier to read, and code that's easy to read is code that's easy to debug:

```
foo = { 'pi' : 3.14,
        'name': 'GAD',
        'age' : 6 }
```

The thing to remember about this being a dictionary is that `foo` has elements and that the elements are referenced by name, not position. In fact, you cannot reference the elements by position, because they are strictly non-ordered. Like the other arrays mentioned, we reference elements with square brackets, only now we need to give the *key* for that element instead of its position:

```
print foo['name']
```

Got it? I hope so, because it's about to get complicated. Python allows data types to be nested, which means that we can create a list of dictionaries or a dictionary full of lists. Hell, we can have a list with a dictionary, a tuple, a string, and a list of dictionaries that contains 19 lists of dictionaries within it! Just because you can doesn't mean you should, though. Still, this has huge benefits as the needs of a script become more complex, and you'll see this nesting with eAPI. The trick to understanding all of this nesting is that each level of nesting is referenced in order from left to right, though it can be easier to think about the references as being from outside-in. Huh? Let's look at an example. Consider this pile of Python:

```
foo = { 'name': {'first': 'Frederick', 'last': 'Smith'},
        'skills': ['singing', 'pilot', 'pirate'],
        'DOB': '02-29-1888',
        'age': 5 }
```

In this example, `foo` is a dictionary with the keys ‘`name`’, ‘`skills`’, ‘`DOB`’, and ‘`age`’. To reference the `age` key within the dictionary `foo`, we use the following:

```
print foo['age']
```

Simple! But what about the first name? Because there’s a dictionary nested within the ‘`name`’ element of `foo` (keep in mind, dictionaries are not ordered—the elements are named), we must reference the elements from the top level down, left to right. Remember, here’s the entire nested array with the data we’re trying to get to in bold:

```
foo = {  'name': {'first': 'Frederick', 'last': 'Smith'},  
        'skills': ['singing', 'pilot', 'pirate'],  
        'DOB': '02-29-1888',  
        'age': 5 }
```

And here’s how we would reference that data:

```
print foo['name']['first']
```

By the way, if you think the age is wrong, it means you don’t get the joke, so please don’t email me the errata. It’s not my fault that Frederick’s nursemaid had a hearing problem.

NOTE

If you still don’t get the joke it’s from the Gilbert and Sullivan Operetta *The Pirates of Penzance*, in which Frederick is determined to have been born on February 29 on a Leap Year and is thus only five years old:

And so, by a simple arithmetical process, you’ll easily discover that even though you’ve lived 21 years, yet, if we go by birthdays, you’re only five, and a little bit over, and, yes, I did the math to figure out a leap year from the reign of Queen Victoria.

This data nesting quickly can get out of hand, and that's not what we're here to learn. I just wanted you to be able to understand what's going on with JSON and eAPI, so let's get back to that. Imagine this output from the CLI again:

```
Arista#sho ver | json
{
    "uptime": 4111.17,
    "modelName": "DCS-7280SR-48C6-M-F",
    "internalVersion": "4.21.1F-9887494.4211F",
    "systemMacAddress": "28:99:3a:be:9f:92",
    "serialNumber": "SSJ17290598",
    "memTotal": 32458980,
    "bootupTimestamp": 1546895161.0,
    "memFree": 30520076,
    "version": "4.21.1F",
    "architecture": "i386",
    "isIntlVersion": false,
    "internalBuildId": "1497e24b-a79b-48e7-a876-43061e109b92",
    "hardwareRevision": "21.05"
}
```

Look at that output with your newfound Python data-type knowledge. Now imagine that all of that is contained with the variable `response`. How might we print the version?

```
print response['version']
```

That's it! The position of the data is irrelevant because it's not an ordered array (list or tuple), and the position of the data in the string is irrelevant because the output shows only the data we care about. Cool! Actually, that's beyond cool because assuming there's an easy way to get to this data programmatically (there is), this means that there is no need for expect scripting *or* screen scraping. Oh, dry the glistening

tear!

NOTE

That's another *Pirates of Penzance* reference in case you spend all your time reading RFCs and don't ever manage to get to the opera.

Now how do we use this in the real world? More importantly, how do we get this output into a script? First, we need to turn on eAPI.

Configuring eAPI

Normally, eAPI communicates over HTTP or HTTPS (default), which facilitates the sending of CLI commands, the output of which is delivered to us in JSON format. To make that happen, we need to configure eAPI on the switch. We do this primarily through the `management api http-commands` section:

```
Arista(config)#management api http-commands
Arista(config-mgmt-api-http-cmds)#
```

Now that we're here, the minimum configuration required is the act of enabling the feature:

```
Arista(config-mgmt-api-http-cmds)#no shutdown
```

The feature will function after we add a username and password, but first let's look at some other options within this section.

By default, the feature uses HTTPS for security. This is wise, and I

don't recommend using HTTP, but if you feel the need, you can change to the insecure clear-text mode with the `protocol http` command. You can even change the port, just as you can with HTTPS:

```
Arista(config-mgmt-api-http-cmds)#protocol ?
  http  Configure HTTP server options
  https Configure HTTPS server options

Arista(config-mgmt-api-http-cmds)#protocol http ?
  port  Specify the TCP port to serve on
  <cr>

Arista(config-mgmt-api-http-cmds)#protocol http port ?
  <1-65535>  TCP port
```

You cannot enable HTTP without first disabling HTTPS:

```
Arista(config-mgmt-api-http-cmds)#protocol http port 80
% Cannot enable HTTP and HTTPS simultaneously
```

To disable HTTP, just negate the `protocol https` command. Then we can add HTTP:

```
Arista(config-mgmt-api-http-cmds)#no protocol https
Arista(config-mgmt-api-http-cmds)#protocol http port 80
```

If you are using HTTPS, it will work without further configuration (though we will need to add a secure username), but if you so desire, you can add your own certificate:

```
Arista(config-mgmt-api-http-cmds)#protocol https certificate
Enter TEXT certificate. Type 'EOF' on its own line to end.

This is not a real certificate

EOF

Enter TEXT private key. Type 'EOF' on its own line to end.
```

```
This is not a real private key
```

```
EOF
```

Note that you do not need to use the web interface. You can also use a couple of nifty features that allow you to connect to the eAPI interface direct from Linux on the switch itself. This is cool because it's faster not adding the web server overhead, and it also allows you to use eAPI without a username and password. How is that safe? When configured in this mode, the only way that eAPI scripts can run is from the switch itself, generally from Bash. To configure eAPI this way, use the `unix-socket` protocol option:

```
Arista(config-mgmt-api-http-cmds)#protocol unix-socket
```

You can also set up the system to use the web interface but respond only from localhost. Note that using this will limit access to port 8080 by default:

```
Arista(config-mgmt-api-http-cmds)#protocol http localhost
```

Let's get this back to a simple state that we know will work for our purposes:

```
Arista(config-mgmt-api-http-cmds)#protocol https
```

We've gone back and forth a lot, so let's exit the mode (thus saving the configuration) and finish up:

```
Arista(config-mgmt-api-http-cmds)#exit
Arista(config)#
```

Earlier, I said that we also needed to add a secure username. eAPI will not function over HTTP/HTTPS without a username with a password configured (unless using localhost or Unix sockets). This is a very good thing, as we'll see, because eAPI allows for a significant amount of control over the switch. Note that this does not need to be a local user and can absolutely be an Authentication, Authorization, and Accounting (AAA) user controlled through RADIUS or TACACS+. Here, I create a local user named `Script` with a password of `Arista`:

```
Arista(config)#username Script secret Arista
```

eAPI is now ready to be used.

To see the status of eAPI on your switch, issue the `show management api http-commands` command from the CLI (yes, you can run this command through eAPI as well):

```
Arista(config)#sho management api http-commands
Enabled: Yes
HTTPS server: running, set to use port 443
HTTP server: shutdown, set to use port 80
Local HTTP server: shutdown, no authentication, set to use port 8080
Unix Socket server: running, no authentication
VRFs: default
Hits: 1
Last hit: 13 seconds ago
Bytes in: 90
Bytes out: 455
Requests: 1
Commands: 1
Duration: 0.219 seconds
SSL Profile: none
FIPS Mode: No
QoS DSCP: 0
Log Level: none
CSP Frame Ancestor: None
TLS Protocols: 1.0 1.1 1.2
User Requests Bytes in Bytes out Last hit
```

```
-----  
  Script      1      90      455      13 seconds ago  
  
URLs  
-----  
Management1 : https://10.0.0.1:443  
Unix Socket : unix:/var/run/command-api.sock
```

In this output, we can see that the feature is enabled, that HTTPS is enabled and HTTP is shutdown, that the ports are set to defaults, the last time it was hit, what username was used, how many times that username was used, and so on.

Finally, watch out for Virtual Routing and Forwarding (VRF) changing how you connect to eAPI. If you've put your management interface into a VRF (see [Chapter 33](#)) like this

```
interface Management1  
  vrf forwarding Manage  
  ip address 10.0.0.1/24
```

and then you enable eAPI, as discussed earlier

```
management api http-commands  
  no shutdown
```

with this configuration you will not be able to connect to eAPI over that management interface, because you will get a **connection refused** message when trying to do so. To make eAPI work over that interface, you'd need to add eAPI into the VRF in use on the management interface:

```
Arista#conf  
Arista(config)#management api http-commands  
Arista(config-mgmt-api-http-cmds)#vrf Manage
```

An added benefit of doing this is the ability to apply an IP access-list to eAPI within the VRF:

```
Arista(config-mgmt-api-http-cmds-vrf-Manage)#ip ?
  access-group  Configure access control list
```

I'm going to keep it simple here and leave it open, though. Note that you must enable eAPI within the VRF. After you do that, you can exit, and you should be able to hit eAPI as you'd expect:

```
Arista(config-mgmt-api-http-cmds-vrf-Manage)#no shut
Arista(config-mgmt-api-http-cmds-vrf-Manage)#exit
Arista(config-mgmt-api-http-cmds)#
```

You can run eAPI in multiple VRFs, too. Here, I have the switch running with eAPI in two VRFs, named Manage and GAD:

```
Student-20(config)#sho run section api
management api http-commands
  no shutdown
  !
  vrf GAD
    no shutdown
  !
  vrf Manage
    no shutdown
```

eAPI Web Interface

So how do we use eAPI? There are two ways. The first way is to hit it with a web browser. Simply point your browser to your switch's management IP address using HTTPS. In my lab, I have a switch configured with a management interface numbered as 10.0.0.1. I would then hit it using the URL `https://10.0.0.1/explorer`. The `/explorer` at the end can be optional depending on what else you have running on the

switch, but I'd add it just to be safe. Note that you might get an untrusted certificate error due to the certificate being self-signed. You will need to acknowledge this on your browser unless you added a valid certificate.

Figure 30-1 shows what I get after entering the username and password. Note that this output changed dramatically around EOS 4.14, so older versions have much different output, but then you shouldn't be running code that old, anyway. The older interface was simpler but also not as powerful.

Simple Request Script Editor

Simple eAPI request editor

This page lets you craft a single eAPI request, and explore the returned JSON. Note that this form creates real eAPI requests, so any configuration you perform will apply to this switch. Don't know where to start? Read the [API overview](#) or try one of these examples: [Check version](#), [Create an ACL](#), [Show virtual router](#), or [View running-config](#)!

API Endpoint Version

Commands

Format Timestamps AutoComplete ExpandAliases

ID Submit POST request

Request Viewer

```
1 "Enter commands above and click 'Submit POST request'"
```

Response Viewer

```
1
```

Figure 30-1. The eAPI request editor

There's a lot of very complicated stuff here, so let's begin with the easy bits. I enter the command `show version` into the text field labeled Commands. After clicking Submit POST Request, the Explorer Response Viewer opens, as shown in Figure 30-2.

ARISTA
Command API

Explorer Overview Command Documentation

Simple Request Script Editor

Simple eAPI request editor

This page lets you craft a single eAPI request, and explore the returned JSON. Note that this form creates real eAPI requests, so any configuration you perform will apply to this switch. Don't know where to start? Read the [API overview](#) or try one of these examples: [Check version](#), [Create an ACL](#), [Show virtual router](#), or [View running-config](#)!

API Endpoint

Version

Commands
1 show version
2

Format

Timestamps

AutoComplete

ExpandAliases

ID

Submit POST request

Request Viewer

```

1 {
2   "jsonrpc": "2.0",
3   "method": "runCmds",
4   "params": {
5     "format": "json",
6     "timestamps": false,
7     "autoComplete": false,
8     "expandAliases": false,
9   },
10  "cmds": [
11    "show version"
12  ],
13  "version": 1
14 },
15 "id": "EapiExplorer-1"
16 }
```

Response Viewer

```

1 {
2   "jsonrpc": "2.0",
3   "id": "EapiExplorer-1",
4   "result": [
5     {
6       "uptime": 5157953.85,
7       "modelName": "DCS-7280SR-48C6-M-F",
8       "internalVersion": "4.21.1F-9887494.4211F",
9       "systemMacAddress": "28:99:3a:06:6c:53",
10      "serialNumber": "JPE16491904",
11      "memTotal": 32458980,
12      "bootupTimestamp": 1549334471,
13      "memFree": 30371268,
14      "version": "4.21.1F",
15      "architecture": "i386",
16      "isIntlVersion": false,
17      "internalBuildId": "1497e24b-a79b-48e7-a876-43061e109b92",
18      "hardwareRevision": "11.03"
19    }
20  ]
21 }
```

Figure 30-2. eAPI Explorer Response Viewer

In the pane on the right, you should see output similar to what I showed you earlier in the chapter.

WARNING

In older revisions of EOS (pre-4.14) commands sent through the eAPI *could not* be abbreviated. If you attempted to send `sho ver`, it would fail. In this example, the command must be the entire string `show version`. There is now an option to AutoComplete, which is a very welcome addition, though using it adds a small bit of complexity to the process, which I'm not going to cover right now.

A quick note about the web page is in order. First, it's awesome, so kudos to the eAPI development team. Second, it contains some of the best documentation ever to come out of Arista, and it's right there on the top, available by clicking the Command Documentation menu item. If you have any deeper questions about eAPI, I recommend you start there because this is a well-documented feature.

Another new feature of the eAPI web interface is the fact that you actually build and test scripts on it (Figure 30-3), though due to the limitations of it being a web-based environment, the scripts that you can build on it are limited to JavaScript.

Simple Request Script Editor

Interactive eAPI scripting environment

This page lets you interactively play with eAPI (a.k.a Command API), using the `Javascript` programming language. To use, write code in the scripting area below and then click "Run Script". You'll then see all requests, responses and script output in the console box below. Note that these script calls are real eAPI requests, so any configuration you perform over the API will apply to this switch.

[Show API documentation](#)**EXAMPLE SCRIPTS**[Print switch address](#)[Show active intfs](#)[Show configured ACLs](#)[Create example ACL](#)[Remove example ACL](#)

```
1 // This script prints out the switch's MAC address
2
3 // Create an EapiClient object...
4 var eapi = new EapiClient();
5 // and form the eAPI request:
6 var request = eapi.runCmds({'version': 1, 'cmds': ['show version']});
7
8 request.done(function(result){
9     // Hooray, the switch replied with data! Since we sent 1 command,
10    // the data we care about is at result[0]. Extract the MAC address
11    // and print it to the console:
12    var macAddr = result[0]["systemMacAddress"];
13    logMessage("The switch's system MAC address is " + macAddr);
14});
```

SCRIPT OUTPUT: Show requests Show responses

Run Script

» Request [id: EapiExplorer-0]: show version

» Response [id: EapiExplorer-0]: show version

The switch's system MAC address is 28:99:3a:06:6c:53

Figure 30-3. eAPI Explorer Script Editor

I know, I said there wouldn't be any JavaScript, but I'm showing this because it's so darn cool. If you're a JavaScript person, this is extra cool, but the main thing I like about this is that it's like a little learning environment that shows you what's happening every step of the way, and it even includes a pile of sample scripts that you can use to learn from.

For me, though, I script my Arista devices using Python, so that's what we're going to do next.

Scripting with eAPI

This web-page feature is nice and all, but it's not very useful in the day-to-day sense. You probably wouldn't use this page to actually control the switch, although you could. This page is there to help you see what the output of a command would be so that you can test your scripts. To that end, let's go ahead and learn the second and more commonly used method of using eAPI, which is scripting.

Let's begin by running the script from Bash on the switch itself. It's easy, and I don't need to do anything special. We need to start with a couple of special lines. First, the line that informs us where to find the interpreter for this script. I'm using Python, so this is what the first line looks like. Yes, I know this can be different, and I know there are decades-long arguments about what it should be. I'm using this format because I'm the guy with the keyboard. I look forward to your angry letters about my lack of `/usr/bin/env` sensibilities.

```
#!/usr/bin/python
```

Next, we need to import the JSON library so that the script can read the output:

```
from jsonrpclib import Server
```

NOTE

If you're running this script from a non-Arista EOS device, you might need to install the JSON libraries. For example, on Ubuntu, I needed to run the following commands:

```
sudo apt-get install python-pip
sudo pip install jsonrpclib
```

The commands would change based on your flavor of Linux. This is not necessary if you're running your script from Bash on your Arista switch (as I am in this example) because the JSON libraries are already included in the EOS build.

Next, I add a line so that I can reference the URL by using the object **switch**. Note the format and the inclusion of the username and password in the URL. Notice also that the URL does not end in */explorer*, but rather */command-api*. Finally, because I'm in Bash on the switch itself, I can use the loopback address:

```
switch = Server( "https://Script:Arista@127.0.0.1/command-api" )
```

If I had configered eAPI to listen on **unix-sockets**, the format of this line would be as follows:

```
switch = Server( "unix:/var/run/command-api.sock" )
```

Here's the same line when configured for **http localhost**:

```
switch = Server( "http://localhost:8080/command-api" )
```

This line sets up what I call the *target*. For a simple script like the one we're building, this is set once and pretty much forgotten. For a complex script that hits multiple switches, there would need to be a target set for each switch, though you could do this through a loop or any other means that you might devise. I show an example of just that in a couple of pages.

The next line actually performs the action of sending the command and collecting the results. The format is important, and we see some variations in a bit. For now, just copy it exactly (feel free to put in other commands to play around):

```
response = switch.runCmds( 1, [ "show version" ] )
```

In this line, `response` is a new variable that will contain the output of the command, `switch` is the object we created in the last line, `runCmds` is a method from the `switch` object (if object-oriented computing hurts your head, feel free to just nod and keep reading), 1 is the version of eAPI, and everything within the square brackets is an array of CLI commands that will be sent to the switch.

Do you remember what Python data type uses square brackets when being created? That is a Python *list*, which means it's an ordered array that can contain multiple values. The fact that it's ordered is important because the commands will be sent in order. Clever, huh?

Finally, a simple line to print out the results:

```
print response
```

I've saved this script in my home directory with the name *ShowVersion.py*. To make it executable, I need to change the file mode:

```
[admin@Arista ~]$ chmod 755 ShowVersion.py
```

WARNING

Remember, anything you put into your home directory on an Arista switch will vanish when you reboot the switch!

Here's the script in its entirety:

```
[admin@Arista ~]$ cat ShowVersion.py
#!/usr/bin/python

from jsonrpclib import Server
switch = Server( "https://Script:Arista@127.0.0.1/command-api" )
response = switch.runCmds( 1, [ "show version" ] )
print response
```

Let's see what we get when it runs:

```
[admin@Arista ~]$ ./ShowVersion.py
[{'modelName': 'DCS-7280SR-48C6-M-F', 'internalVersion': '4.20.1F-6820520.4201F', 'systemMacAddress': '28:99:3a:be:a0:ba', 'serialNumber': 'SSJ17290599', 'memTotal': 32459704, 'bootupTimestamp': 1525109082.15, 'memFree': 30229568, 'version': '4.20.1F', 'architecture': 'i386', 'isIntlVersion': False, 'internalBuildId': '9a79f1b5-d296-4478-b127-a31835922e5e', 'hardwareRevision': '21.05'}]
```

Well, that doesn't look very useful, but remember, this is the raw JSON output.

WARNING

The behavior of hitting HTTPS targets via Python has changed and is different when using Python locally on EOS 4.21.1F and later.

If you’re playing along at home (or work) and doing this on an Arista switch running EOS 4.21.0F or later, running this script will result in a traceback complaining thusly:

```
[-- lots of ugly stuff removed --]  
[SSL: CERTIFICATE_VERIFY_FAILED] certificate verify failed
```

This is a result of EOS 4.21.1F now using Python version 2.7.14, which is enforcing the idea that self-signed certificates are bad. Because EOS uses a self-signed certificate by default, this simple script will no longer work on post EOS-4.21.0F code. This also means that if you’re running your script from a server or client that uses a newer version of Python, you’ll get the same results. Here are some options to get it working again:

Add a valid certificate in EOS

This is probably the right way to go but seems an awful lot like real work to me. Plus, I probably need to talk to the security team, and that always ends with me wishing I’d never done that.

Instruct Python to allow self-signed certs

This has the benefit of working locally and remotely but is probably frowned upon in today’s “Certificate Authorities need more money” world. In the Python script, add the following lines just after the shebang:

```
import ssl
```

```
ssl._https_verify_certificates( False )
```

Note that in the strictest security sense, this is probably a bad idea, but because we're running the script locally, this is not a big deal. This is a bigger deal if you're accessing the switch remotely, but I'll leave it up to you to decide how bad it is to use self-signed certificates.

Enable `http localhost` in the eAPI configuration

Note that this will bypass encryption altogether, but will work only when running the script from the switch itself. In EOS, configure the following within the `management api http-commands` section:

```
Arista(config-mgmt-api-http-cmds)#protocol http local
```

Then, change the target line in the script to the following:

```
switch = Server( "http://127.0.0.1:8080/command-api" )
```

Enable `unix sockets` in the eAPI configuration

This, too, will work only when running scripts from localhost. In EOS, add the following in the `management api http-commands` section:

```
Arista(config-mgmt-api-http-cmds)#protocol unix-socket
```

Then change the target line in the script to the following:

```
switch = Server( "unix:/var/run/command-api.sock" )
```

For each of these solutions, the only things that change are the lines listed in each example. The rest of the script should not need to change at all.

Let's add some lines to our script to make it more useful. Remember how I showed that this jumble of text was actually a set of key–value pairs? Here's an easy way to print that same output in an easy-to-read manner similar to what's seen in the CLI with the `| json` modifier: use the Python pretty-print library.

```
#!/usr/bin/python

import pprint

from jsonrpclib import Server
switch = Server( "https://Script:Arista@127.0.0.1/command-api" )
response = switch.runCmds( 1, [ "show version" ] )
pprint.pprint(response)
```

By adding the `import pprint` line in the beginning and changing `print response` to `pprint.pprint(response)`, we now get a more readable output:

```
[admin@Arista ~]$ ./ShowVersion.py
[{'architecture': 'i386',
 'bootupTimestamp': 1546976871.0,
 'hardwareRevision': '21.05',
 'internalBuildId': '35b4398c-100e-45ff-aae4-857ac15fbff4',
 'internalVersion': '4.21.2F-10430819.4212F',
 'isIntlVersion': False,
 'memFree': 30434720,
 'memTotal': 32458984,
 'modelName': 'DCS-7280SR-48C6-M-F',
 'serialNumber': 'SSJ17290598',
 'systemMacAddress': '28:99:3a:be:9f:92',
 'uptime': 8896.37,
 'version': '4.21.2F'}]
```

One of the nice things about tools like `pprint` is that the output is sorted. Dictionaries in Python have no order, but sorting the keys can make it easier to find them for us pathetic humans.

NOTE

If you're wondering why the new line says `pprint.pprint(response)` instead of just `pprint(response)`, that's because we're referencing the function `pprint()` from within the `pprint` module. If the module had been called *gad*, the line would say `gad.pprint(response)`.

There is another option for showing the output of JSON-encoded data in Python, and that's using the `json` library:

```
#!/usr/bin/python
import json
from jsonrpclib import Server
switch = Server( "unix:/var/run/command-api.sock" )
response = switch.runCmds( 1, [ "show version" ] )
print json.dumps(response, indent=3)
```

This will result in a similar but more configurable output that lets you do things like change the number of spaces used for indents:

```
[admin@Arista ~]$ ./ShowVersion.py
[
  {
    "uptime": 8594.43,
    "modelName": "DCS-7280SR-48C6-M-F",
    "internalVersion": "4.21.2F-10430819.4212F",
    "systemMacAddress": "28:99:3a:be:9f:92",
    "serialNumber": "SSJ17290598",
    "memTotal": 32458984,
    "bootupTimestamp": 1546976871.0,
    "memFree": 30434612,
    "version": "4.21.2F",
    "architecture": "i386",
    "isIntlVersion": false,
    "internalBuildId": "35b4398c-100e-45ff-aae4-857ac15fbff4",
    "hardwareRevision": "21.05"
  }
]
```

Regardless of how you print, the entire output is stored in the variable `response`, but more specifically, we can pull out keys and values within `response`. For example, the version has the key name of (surprisingly) `version`. To reference it, I would write the following line, replacing the simple `print response` line:

```
print response[0]['version']
```

Let's examine that for a second. First, I went back to `print` because I only needed pretty-print for the full output and that's rarely needed when referencing single elements. Second, why does the line have `response[0]` in it? Isn't the output a dictionary? It is, but if you look at the full output, the dictionary is within a list—note the square brackets at the beginning and end. This is because the output includes the output of every command (sent as a dictionary) with each of those dictionaries sent as an element within a list. The list is the output of each command you sent in the order in which you sent the commands while the dictionaries within each element are the outputs of those commands. Because we sent only one command, the output is in element `[0]`. Within element zero (the output of the command `show version`), we want the key named `version`. Thus:

```
print response[0]['version']
```

Now let's see what we get when we run the script:

```
[admin@Arista ~]$ ./ShowVersion.py
4.21.1F
```

That's certainly more manageable, but let's dress it up a bit, like this:

```
print "The system version is:      ", response[0][ "version" ]
```

If you're not up on your Python (specifically Python 2.7), this is printing a literal string followed by the output from the `show version` command.

Now when we run the script, we get the following, much more useful result:

```
The system version is:      4.21.1F
```

Let's pull out some more information. Let's add the following lines:

```
print
print "The system MAC address is:  ", response[0][ "systemMacAddress" ]
print "The system version is:      ", response[0][ "version" ]
print "The system architecture is: ", response[0][ "architecture" ]
```

Now when we run the script, we get the following results:

```
[admin@Arista ~]$ ./ShowVersion.py
```

```
The system MAC address is:  28:99:3a:be:a0:ba
The system version is:      4.21.1F
The system architecture is: i386
```

If you're wondering where I got the key names (`systemMacAddress`, `version`, `architecture`), they are all keys in the preceding JSON output. Take a look again and see for yourself. Also, this is where the web page comes in handy because it shows you all the keys and their values without having to write a script to figure out what they are (`command | json` in CLI is even better!). Finally, watch out for camel case. Look at the key names and pay attention to the capital letters in the middle of some of them. They matter, so if your script isn't

working, check there first.

Here's the final script:

```
[admin@Arista ~]$ cat ShowVersion.py
#!/usr/bin/python

from jsonrpclib import Server
switch = Server( "https://Script:Arista@127.0.0.1/command-api" )
response = switch.runCmds( 1, [ "show version" ] )

print
print "The system MAC address is:  ", response[0][ "systemMacAddress" ]
print "The system version is:      ", response[0][ "version" ]
print "The system architecture is: ", response[0][ "architecture" ]
```

I could do this from any machine that can reach the switch using HTTPS (or HTTP if so configured). Think about that for a minute: I could write scripts on my Mac at home that could control my Arista switches at work! Of course, I should need to go through a VPN or have some other sort of security in place, but the fact remains that I could absolutely control remote switches from machines elsewhere. In my training classes I control 24 switches with eAPI scripts just like this one. Let's see how.

Suppose that I have a rack full of switches and they're numbered 10.0.0.1 through 10.0.0.10 and I want to poll them from a server. To issue the same command on all of them, I alter my eAPI script and put in a loop:

```
#!/usr/bin/python

from jsonrpclib import Server

print "Switch#  System Mac          Version  Architecture"
print "-----"
```

```

for x in range(1,11):
    x = str(x)

    switch = Server( "https://Script:Arista@10.0.0." + x + \
                    "/command-api" )
    response = switch.runCmds( 1, [ "show version" ] )

    print "Switch-" + x.ljust(3) + \
          response[0][ "systemMacAddress" ].ljust(19) + \
          response[0][ "version" ].ljust(9) + \
          response[0][ "architecture" ]

```

I've added some lines, most of which have to do with formatting:

```

print "Switch#  System Mac          Version  Architecture"
print "-----"

```

All they do is print a header. The next two lines create a loop that iterates through the range of 1 to 10. The reason it looks like it's 1 to 11 is because Python counts from 0, so the `range(1,11)` statement is actually saying "create a list of 11 numbers (0 to 10) and start at 1. Thus, 1–10." The line after the `for` statement just converts the number from an integer to a string because we'll be concatenating to a string later on, and you cannot concatenate an integer and a string:

```

for x in range(1,11):
    x = str(x)

```

The next line has been altered so that the variable `x` is now injected into the fourth octet of the IP address. This will have the result of the IP address being 10.0.0.1, 10.0.0.2, and so on, until we get up to 10.0.0.10:

```

switch = Server( "https://Script:Arista@10.0.0." + x + \
                  "/command-api" )

```

Finally, the last few lines look like this:

```
response = switch.runCmds( 1, [ "show version" ] )

print "Switch-" + x.ljust(3) +
      response[0][ "systemMacAddress" ].ljust(19) + \
      response[0][ "version" ].ljust(9) + \
      response[0][ "architecture" ]
```

The first line in that block is the same as our previous example. The last line (those four physical lines of code are one logical line thanks to the backslash line-continuation character at the end of the first three) just prints each of the fields: `systemMacAddress` (left justified in 19 spaces), `version` (left justified in 9 spaces, and `architecture` (no justification).

Here's what that script looks like when it runs:

```
gad@[ALab]:~/eAPI$ ./eAPI-Loop.py
Switch#  System Mac          Version  Architecture
-----
Switch-1 28:99:3a:be:9f:92  4.20.3F  i386
Switch-2 28:99:3a:be:9d:d6  4.20.3F  i386
Switch-3 28:99:3a:be:9e:20  4.20.1F  i386
Switch-4 28:99:3a:be:a0:70  4.20.1F  i386
Switch-5 28:99:3a:be:a1:04  4.20.3F  i386
Switch-6 28:99:3a:be:9f:48  4.20.3F  i386
Switch-7 28:99:3a:be:9c:1a  4.20.3F  i386
Switch-8 28:99:3a:be:9c:ae  4.20.3F  i386
Switch-9 28:99:3a:be:9b:3c  4.20.3F  i386
Switch-10 28:99:3a:be:9a:5e 4.20.3F  i386
```

Because I put the commands into a loop, I've hit 10 switches and retrieved some useful (and some not-so-useful) information from each and then reported it all into a nice concise report.

But what about commands that require additional input? For example,

if I want to run a privileged command? Here's an example:

```
#!/usr/bin/python

from jsonrpclib import Server

switch = Server( "https://Script:Arista@10.0.0.1/command-api" )

response = switch.runCmds( 1, [ "reload now" ] )
```

Everything we've done so far you can do without entering the `enable` command, but if we try to issue the `reload now` command (which requires a privilege level higher than the default level of 1) via the eAPI, we get the following:

```
[GAD@Server ~]$ ./SimpleReloadNow.py
Traceback (most recent call last):
  File "./SimpleReloadNow.py", line 7, in <module>
    response = switch.runCmds( 1, [ "reload now" ] )
  File "/usr/local/lib/python2.7/dist-packages/jsonrpclib/jsonrpc.py",
line 288, in __call__
    return self.__send(self.__name, args)
  File "/usr/local/lib/python2.7/dist-packages/jsonrpclib/jsonrpc.py",
line 238, in _request
    check_for_errors(response)
  File "/usr/local/lib/python2.7/dist-packages/jsonrpclib/jsonrpc.py",
line 567, in check_for_errors
    raise ProtocolError((code, message))
jsonrpclib.jsonrpc.ProtocolError: (1002, u"CLI command 1 of 1
'reload now' failed: invalid command")
```

The last line indicates that we've issued an invalid command because we didn't have the proper privilege level to execute the `reload now` command (it's invalid because it doesn't exist in `disabled exec` mode). To issue the `reload now` command, we first need to enter `enable mode`. To do that within a script, simply stack the commands into a list within the `runCmds` line like this:

```
response = switch.runCmds( 1 , [ "enable" , "reload now" ] )
```

Now when I execute this from Bash on the switch, I get the following output:

```
[GAD@Arista ~]$ ./ReloadNow.py
Broadcast message from root@Arista
(unknown) at 11:01 ...
The system is going down for reboot NOW!
```

Note that on some versions of code (circa 2018), running this script remotely will result in an odd traceback error:

```
gad@[ALab]:~/eAPI$ ./SimpleReloadNow.py
Traceback (most recent call last):
  File "./SimpleReloadNow.py", line 8, in <module>
    response = switch.runCmds( 1, [ "enable" , "reload now" ] )
  File "/usr/local/lib/python2.7/dist-packages/jsonrpclib/jsonrpclib.py",
line 288, in __call__
    return self.__send(self.__name, args)
  File "/usr/local/lib/python2.7/dist-packages/jsonrpclib/jsonrpclib.py",
line 237, in _request
    response = self._run_request(request)
  File "/usr/local/lib/python2.7/dist-packages/jsonrpclib/jsonrpclib.py",
line 255, in _run_request
    verbose=self.__verbose
  File "/usr/lib/python2.7/xmlrpclib.py", line 1273, in request
    return self.single_request(host, handler, request_body, verbose)
  File "/usr/lib/python2.7/xmlrpclib.py", line 1306, in single_request
    return self.parse_response(response)
  File "/usr/lib/python2.7/xmlrpclib.py", line 1462, in parse_response
    stream = GzipDecodedResponse(response)
  File "/usr/lib/python2.7/xmlrpclib.py", line 1213, in __init__
    self.stringio = StringIO.StringIO(response.read())
  File "/usr/lib/python2.7/httplib.py", line 549, in read
    return self._read_chunked(amt)
  File "/usr/lib/python2.7/httplib.py", line 603, in _read_chunked
    raise IncompleteRead(''.join(value))
httplib.IncompleteRead: IncompleteRead(50 bytes read)
```

On older code I would simply not get any output because the `reload now` command does not offer any output to print, but something changed and now the switch reloads before eAPI can send a response back, so the script pukes. You can handle this with exception handling, but that's kind of outside the scope of this book.

This is a fabulous feature, but be aware of the danger of leaving it open, especially if, like far too many installations I've encountered, you have a well-known username and password on your devices. Why? Here's a script that reboots a switch. The script is written such that the IP address of the switch is entered at the command line:

```
#!/usr/bin/python

import sys
from jsonrpclib import Server
target = "https://Script:Arista@" + str(sys.argv[1]) + "/command-api"
switch = Server(target)

response = switch.runCmds( 1, [ "enable" , "reload now" ] )
print "Response:", response
```

Let's assume that I have enabled eAPI on all of my switches and that I have enabled the username `Script` on all of my switches, as well. I wrote the previous script on my Linux box, and I named it `RebootNow.py`. From my Unix box, I type `./RebootNow.py 10.0.0.1` and the switch at 10.0.0.1 reboots. I could just as easily write a loop so that my little script reboots 100 (or 1,000) switches at a time. Yeah, don't do that. Note that this script works only if the switch has no *enable secret* password. More on that in a minute.

Note that you cannot use interactive commands, so things like `top`, `watch`, and anything that asks for input will not work. eAPI is not an

interactive tool; you must send all of the commands needed to accomplish something in one session. It's easy because those commands are sent in a list. Here's an example of a script that changes the description on Ethernet 5. Note that all of the commands are sent at once:

```
#!/usr/bin/python

from jsonrpclib import Server
switch = Server( "https://Script:Arista@127.0.0.1/command-api" )
response = switch.runCmds( 1, [ "enable" ,
                                "configure" ,
                                "interface ethernet 5" ,
                                "description I like pie!" ] )
print response
```

When I run this script, I get this weird output:

```
[GAD@Arista ~]$ ./ILikePie.py
[{}, {}, {}, {}]
```

What's up with that? We sent four commands, so there is a list (square brackets) that contains four dictionaries (curly braces) in the order in which we sent the commands. The four commands we sent produce no output, though, so each of those dictionaries is empty. Let's see if it worked (the script would have thrown an error if it did not). Note also that we're running this script locally on an Arista switch, which is why this next command works:

```
[GAD@Arista ~]$ Cli -p15 -c "sho run int e5"
interface Ethernet5
    description I like pie!
```

Nice!

I mentioned that interaction is not allowed, and that's why I used `reload now` instead of the normal `reload` command—`reload` is interactive; therefore, it asks, “are you sure?” whereas `reload now` is not. Sometimes, though, you need to interact. Consider the `enable` command. If you need to enable like we did in the last example, what happens if there's an `enable secret` password?

In this case, the `enable` command has an eAPI non-interactive mode. To use it, we send the command within a dictionary with two keys, like this:

```
#!/usr/bin/python
from jsonrpclib import Server
switch = Server( "https://Script:Arista@127.0.0.1/command-api" )
response = switch.runCmds( 1, [ { "cmd": "enable",
                                  "input": "SuperSecret" } ,
                                  "configure" ,
                                  "interface ethernet 5" ,
                                  "description GAD" ] )
print response
```

Let's run it:

```
[admin@Arista ~]$ ./ILikePie.py
[{'messages': ['Warning: Password input may be echoed.\nPassword:
\n']}, {}, {}, {}]
```

This is an example of a warning message being returned via eAPI. Note that the command still worked:

```
[admin@Arista ~]$ Cli -p15 -c "sho run int e5"
interface Ethernet5
  description GAD
```

A warning message is a noncritical message that you might see if you

entered the command from the CLI. Managing these messages is outside the scope of this book, but if you'd like to learn more, I suggest reading the excellent documentation or taking an Arista programming class. Even better, convince O'Reilly that the world needs an Arista network programming book!

The last quick-and-dirty example I include is the ability to save the *running-config*. Although you can output the *running-config* in JSON format, I find that to not be terribly useful, because you can't use that output to reconfigure a switch (at least not as of this writing). To get the output of a command as you would see it on the CLI, use the *text* mode. Normally, eAPI outputs in **json** mode, and because it's the default, I haven't included it in my scripts. If I did, it would look like this:

Changing that field to text gives us this script:

Note that the “text” string is outside of the list containing the commands being sent. This changes the output to simple text mode:

```
[admin@Arista ~]$ ./ShowRun.py
[{'output': 'Warning: Password input may be echoed.\nPassword: \n'}, 
 {'output': '! Command: show running-config\n! device: Arista
(DCS-7280SE-72, EOS-4.18.3.1F)\n!\n! boot system flash:/EOS-
4.18.3.1F.swi\n!\nalias conint sh interface | i connected\nalias
senz show interface counter error | nz\nalias shmc show int | awk
'^[A-Z]/ { intf = $1 } /, address is/ { print intf, $6 }'\nalias
snz show interface counter | nz\nalias spd show port-channel %1
detail all\nalias sqnz show interface counter queue | nz\nalias
srnz show interface counter rate | nz\n!\nalias intdesc\n  !!
Usage: intdesc interface-name description\n  [--output truncated--]
```

What the heck? Oh yeah, how many commands did we send? Two. Even though the output is in text mode, the output is still in a list and the output of the `show running-config` command is in element [1]. Note also that each element contains a single key with the name *output*. The output is still a dictionary for each command in a list; it’s just that we need to reference the right element and then the *output* key. Let’s fix that:

```
#!/usr/bin/python

from jsonrpclib import Server
switch = Server( "https://Script:Arista@127.0.0.1/command-api" )
response = switch.runCmds( 1, [ { "cmd": "enable",
                                 "input": "SuperSecret" } ,
                                 "show running-config" ] ,
                                 "text" )
print response[1]["output"]
```

Now, when we run the script, we get this:

```
[GAD@Arista ~]$ ./ShowRun.py
! Command: show running-config
! device: Arista (DCS-7280SE-72, EOS-4.18.3.1F)
!
```

```
! boot system flash:/EOS-4.18.3.1F.swi
!
alias conint sh interface | i connected
alias senz show interface counter error | nz
alias shmc show int | awk '/^[A-Z]/ { intf = $1 } /, address is/
{ print intf, $6 }'
alias snz show interface counter | nz
alias spd show port-channel %1 detail all
alias sqnz show interface counter queue | nz
alias srnz show interface counter rate | nz
!
alias intdesc
  !! Usage: intdesc interface-name description
  10 config
  20 int %1
  30 desc %2
  40 exit
[--output truncated--]
```

Nice! This easily could be converted to a script that writes that output to a file with a timestamp.

Conclusion

eAPI is a tremendously powerful tool, and honestly, I could write an entire book on it, but in the interest of space I need to stop here so that this chapter doesn't overwhelm this book. (Tell O'Reilly that you want a book on Arista eAPI, and I'll write one!) There are a whole host of topics that can make your scripts better, from error handling at the network level to dealing with eAPI errors to just making your scripts useful with easy to read output. I heartily recommend that you just play around with this feature (though not on a live switch!) to see just how powerful it can be. I easily automate hundreds of switches with eAPI, and you can, too.

You know what else is cool about eAPI? The software tool

CloudVision Portal uses eAPI to communicate with Arista switches. All the cool stuff that CloudVision does in regard to sending and validating configurations, updating switch configurations, and just about everything else that involves communicating with the physical switches is done through eAPI. It's that powerful.

Chapter 31. Containers

Containers are a modern take on the virtual machine idea where the base operating system (OS) of a host computer provides the kernel-level needs for the container. In a traditional virtual machine (VM), each VM is a complete standalone system with its own kernel and simulated hardware. In a container environment, the kernel doesn't need to be loaded because it's already there thanks to the kernel being shared with the host operating system. This also means that the simulation of hardware is not required, which significantly reduces the overhead necessary to run a container. For these reasons, containers tend to be smaller and faster and, perhaps most important, they can start almost instantly.

Where are containers a big deal? Imagine a website that serves millions of users. During times of high utilization more VMs might have been added to deal with the increased load, but VMs are expensive and slow to deploy. With containers, Apache could be loaded into a container and started almost instantly and with much lower overhead than a VM. Thousands of Apache containers could be spun up in a matter of seconds with much lower cost than an equivalent VM rollout.

Finally, another benefit of containers is that an application can be bundled together with all of its dependencies so that an installation process doesn't need to be performed in order to run an application. Because the containers are self-contained, the same container can run

on a Mac, on Linux, on Windows, or even an Arista switch, assuming there are memory resources available to support the container.

Although containers can be used to spin up a quick instance of an application, they can also be used similarly to a VM. Containerized EOS, or cEOS, is an example of using a container like a VM, and I'm going to show you a particularly wacky way of doing that while also explaining why cEOS has had a fairly large impact on some pretty large data center infrastructures.

Why EOS Containers?

If you're wondering why you would want to do this when you've got perfectly good VMs running, there are a couple of reasons such as decreased CPU and memory utilization, but that's not really what cEOS is about.

cEOS allows Arista to put its killer Network Operation System (NOS) into switches that aren't built by Arista. This concept is commonly called white-box switching where the hardware is largely commoditized and the NOS is what matters. Some very large cloud titans are doing things with white-box switching and would like to use Arista's EOS in that environment because they know how great EOS is.

Remember, the forwarding of packets on a modern switch is done by the Application-Specific Integrated Circuit (ASIC), and those ASICs are off-the-shelf parts on many vendors' switches thanks in large part to Arista championing the idea of doing so. If every vendor is making a

Jericho-based switch, why not buy a generic Jericho-based switch and put Arista's great EOS on it? A great switch is more than just its ASIC, and EOS is a huge part of what makes Arista switches great. Because EOS's job is to control the ASIC, a white-box switch running Linux can now also run EOS.

The cloud titans often control millions of containers with something called an *orchestrator*. With cEOS in the mix, the Top-of-Rack (ToR) switches can also be built and managed by the same orchestrator used for the other containers. That's the kind of thing that can vastly simplify these massive environments, and simplicity in a massive environment is always a good thing.

When it comes to containers, there are two ways that EOS can be involved. EOS can be run in a container, and containers can be run in EOS. Let's take a look at each scenario.

cEOS—EOS in a Container

Docker has many versions and types, but for my Mac, I'm going to install Docker Desktop. As of early 2019, this is roughly a 500 MB download. Read those limitations on the page! I had to remove the old version of Virtual Box from my system in order to make Docker work. When the download finished, I ran the image and was treated to the delightful install instruction page shown in Figure 31-1. There are even waves on the bottom!

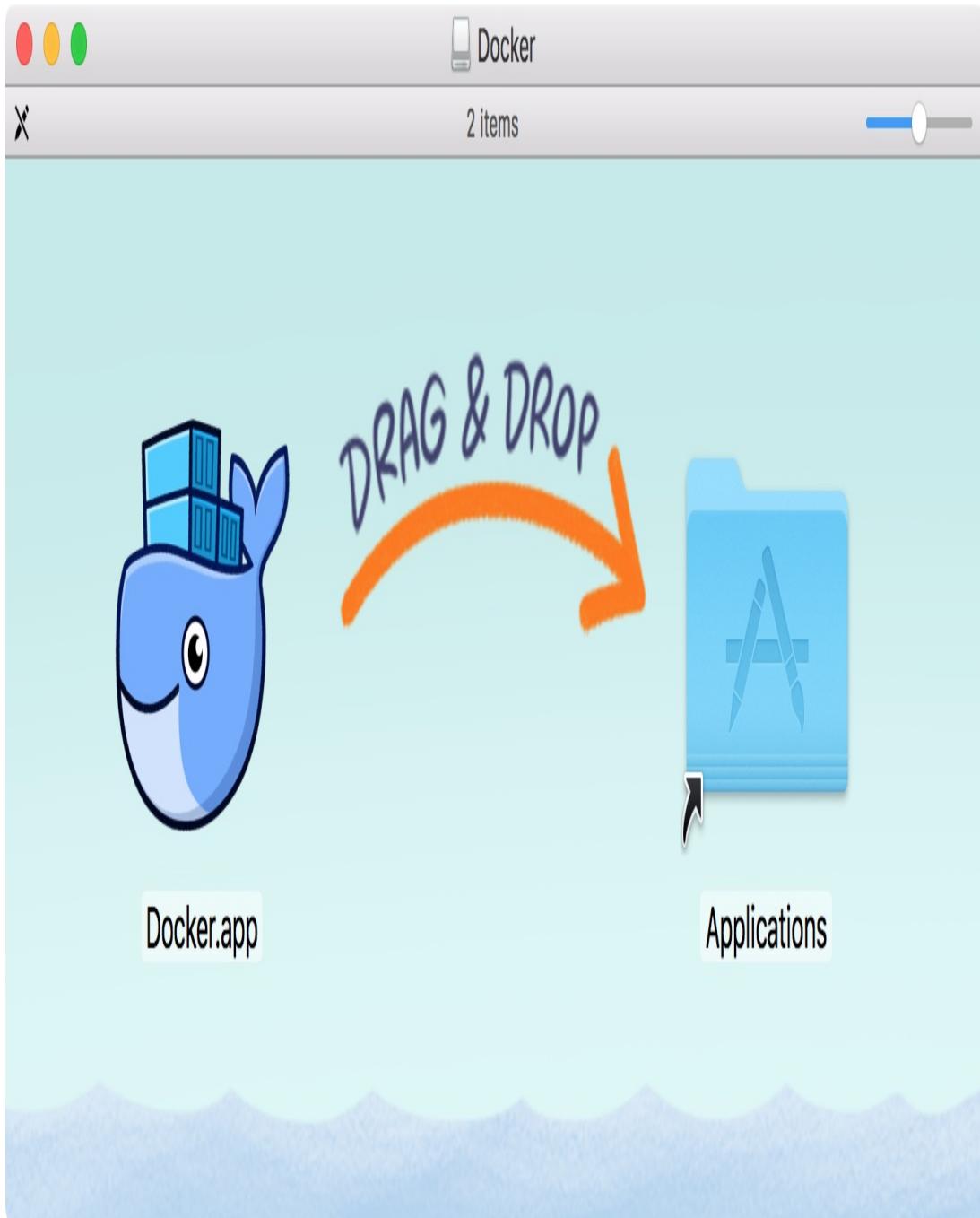


Figure 31-1. The Docker installer screen

After Docker is installed, you'll need to log in to make it work using the same username and password that you created to download the application. When it's working, you should have the new command `docker` available in your terminal app. The first thing to try is the

`docker run hello-world` command to see whether it's working. You should see something similar to the following:

```
GAD-15-Pro:~ gad$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
1b930d010525: Pull complete
Digest: sha256:2557e3c07ed1e38f26e389462d03[...]21577a99efb77324b0fe535
Status: Downloaded newer image for hello-world:latest
```

Hello from Docker!

This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:

1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub. (amd64)
3. The Docker daemon created a new container from that image which runs the executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
`$ docker run -it ubuntu bash`

Share images, automate workflows, and more with a free Docker ID:
<https://hub.docker.com/>

For more examples and ideas, visit:
<https://docs.docker.com/get-started/>

```
GAD-15-Pro:~ gad$
```

Nice! OK, let's get some cEOS action going! To get this working, you need to go to the [Arista software downloads page](#) (non-guest user/password required). I grabbed the latest cEOS-Lab image (4.21.1F) and then did the following:

```
GAD-15-Pro:~ gad$ mkdir Docker
GAD-15-Pro:~ gad$ cd Docker/
```

```
GAD-15-Pro:Docker gad$ mv Downloads/cEOS-lab.tar.xz .
```

This places the downloaded cEOS image into a Docker directory in my *homedir*. After that's done, I import the container into Docker. This took the better part of a minute on my 2016 i7 Macbook Pro (I refuse to upgrade to the new Macbook Pros because the keyboards are horrible):

```
GAD-15-Pro:Docker gad$ docker import cEOS-lab.tar.xz
ceosimage:4.21.1F
sha256:cd5cce52a6866a4f12c99992135ad18c24eb47d0c9[...]ad54d5666c9461eb2f
```

To prove that it's there, you can use the `docker images` command:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ceosimage	4.21.1F	cd5cce52a686	2 minutes ago	1.5GB
hello-world	latest	fce289e99eb9	5 weeks ago	1.84kB

After reading the readme document that comes with cEOS, I pretty much just copied this command from there right into my terminal window:

```
GAD-15-Pro:Docker gad$ docker create --name=ceos --privileged
-p 443:443 -e CEOS=1 -e container=docker -e
EOS_PLATFORM=ceoslab
-e SKIP_ZEROTOUCH_BARRIER_IN_SYSDBINIT=1 -e ETBA=1
-e INTFTYPE=eth -it ceosimage:4.21.1F /sbin/init
```

That's a long ugly command line, so I reformatted it and put it into a script I called `create_ceos`:

```
#!/bin/bash
docker create --name=ceos
  --privileged
  -p 443:443
  -e CEOS=1
```

```
-e ETBA=1
-e INTFTYPE=eth
-e container=docker
-e EOS_PLATFORM=ceoslab
-e SKIP_ZEROTOUCH_BARRIER_IN_SYSDBINIT=1
-it ceosimage:4.21.1F
/sbin/init
```

I changed it to have exec privileges so that I could just run it:

```
GAD-15-Pro:Docker gad$ chmod 755 create_ceos.sh
GAD-15-Pro:Docker gad$ ./create_ceos.sh
65b09f9a54a2104b776ca19e1af677497bdbf344f7f63f011c29458baceed360
```

By the way, if you see an error like this, it means you have spaces or tabs after the backslashes:

```
GAD-15-Pro:Docker gad$ ./create_ceos.sh
invalid reference format
./launch_ceos.sh: line 6: -e: command not found
./launch_ceos.sh: line 9: -e: command not found
./launch_ceos.sh: line 10: -e: command not found
```

NOTE

A cool way to see spurious trailing whitespace in vi is with the `:set list` command, which shows all of the newlines as dollar signs (among other things).

Those options are set based on the readme file in the software downloads section for the version of cEOS you downloaded. Here's what they all do:

`--name=ceos`

(Docker flag) Creates a container with the name ceos. You can adjust to taste.

--privileged
(Docker flag) Runs the container in privileged mode.

-p 443:443
(Docker flag) Exposes port 443, which we might use for eAPI.

-e CEOS=1
(cEOS flag) Informs EOS that this is cEOS and not “regular” EOS.

-e ETBA=1
(cEOS flag) Instructs EOS to use a software-based data-plane.

-e INTFTYPE=eth
(cEOS flag) Instructs the underlying OS to label the interfaces as eth.

-e container=docker
(cEOS flag) There are other container types than Docker. This informs cEOS that it's in a Docker container.

-e EOS_PLATFORM=ceoslab
(cEOS flag) Check the readme for the cEOS version you've downloaded to see what this should be set to because different versions have different settings.

-e SKIP_ZEROTOUCH_BARRIER_IN_SYSDBINIT=1
(cEOS flag) Instructs EOS to not run Zero-Touch Provisioning (ZTP) on boot.

-it ceosimage:EOS-4.21.1F
(Docker flag) Specifies to Docker which image to use (this should reflect what you called it when you imported it).

/sbin/init
(Docker flag) This is the command that you use to start the container.

With all of that covered, let's see what Docker thinks is going on after we ran that big command:

```
GAD-15-Pro:Docker gad$ docker ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS NAMES
842bec7b060e ceosimage:4.21.1F "/sbin/init" 23 min ago Created ceos
621b5ad61906 hello-world "/hello" 32 min ago Exited ac_new
```

Just a note that this command creates a ridiculously wide output that I had to squash considerably to make it fit on this page without wrapping. I had to remove a column (ports) that had nothing in it, and I had to compress the names and alter some of the output, but none of that really affects what we're doing.

Now let's create a couple of quick and dirty networks:

```
GAD-15-Pro:Docker gad$ docker network create net-1
a7c57b81276d67d7b77d2d0e31b86c929f18c00f55536db96b75ebf1d1d6417b
GAD-15-Pro:Docker gad$ docker network create net-2
0969f62f7fe8028583301ccba6bdcb49276770a3e2b82d08a30c78f8d1de2662
```

With those created, we need to add them to the container:

```
GAD-15-Pro:Docker gad$ docker network connect net-1 ceos
GAD-15-Pro:Docker gad$ docker network connect net-2 ceos
```

Time to start that container:

```
GAD-15-Pro:Docker gad$ docker start ceos
ceos
```

To see whether it's running, use the `docker ps` command:

```
GAD-15-Pro:Docker gad$ docker ps
CONTAINER ID IMAGE CREATED STATUS NAMES
842bec7b060e ceosimage:4.21.1F 33 minutes ago Up ceos
```

This also created a super-wide output in which I had to remove the following columns:

```
COMMAND      PORTS
"/sbin/init" 0.0.0.0:443->443/tcp
```

It's almost like software developers are writing code for their super-cool wide screens and not taking into consideration lonely writers sitting in Starbucks trying to make their output fit into the width of a book.

At any rate, with the container running, we should be able to attach to it. If you use the `docker attach ceos` command, you'll see it booting, but you won't get a command-line interface (CLI) prompt. To interact with the container, use the `docker exec -it containernname Cli` command:

```
GAD-15-Pro:Docker gad$ docker exec -it ceos Cli
localhost>en
localhost#sho int status
Port      Name    Status      Vlan      Duplex  Speed  Type[...]
Et1          connected    1        full    unconf EbraTestPhyPort
Et2          connected    1        full    unconf EbraTestPhyPort
```

Typing `exit` in the container returns you to your OS:

```
localhost#exit
GAD-15-Pro:Docker gad$
```

Coolness! To stop the container, use the `docker stop containernname` command:

```
GAD-15-Pro:Docker gad$ docker stop ceos
```

How about making two of them that can network with each other? First, get rid of the container we just made:

```
GAD-15-Pro:Docker gad$ docker container rm ceos
ceos
```

And the networks:

```
GAD-15-Pro:Docker gad$ docker network rm net-1
net-1
GAD-15-Pro:Docker gad$ docker network rm net-2
net-2
```

Here's the updated script, which creates two containers named cEOS-1 and cEOS-2 along with net-1 and net-2 networks, and then connects them. I've removed the port connection to 443 because I'm not using it here and I'd need to map each of the containers differently:

```
#!/bin/bash
docker create --name=cEOS-1 \
    --privileged \
    -e CEOS=1 \
    -e container=docker \
    -e EOS_PLATFORM=ceoslab \
    -e SKIP_ZEROTOUCH_BARRIER_IN_SYSDBINIT=1 \
    -e ETBA=1 \
    -e INTFTYPE=eth \
    -it ceosimage:4.21.1F \
    /sbin/init

docker create --name=cEOS-2 \
    --privileged \
    -e CEOS=1 \
    -e container=docker \
    -e EOS_PLATFORM=ceoslab \
    -e SKIP_ZEROTOUCH_BARRIER_IN_SYSDBINIT=1 \
    -e ETBA=1 \
    -e INTFTYPE=eth \
    -it ceosimage:4.21.1F \
    /sbin/init

docker network create net-1
docker network create net-2

docker network connect net-1 cEOS-1
```

```
docker network connect net-2 cEOS-1  
docker network connect net-1 cEOS-2  
docker network connect net-2 cEOS-2
```

Run the script:

```
GAD-15-Pro:Docker gad$ ./create_ceos.sh  
7d612e967d7ef1495882ec76457b1718e73a2542b848552f9f12201aa33af13  
fa07ea04ebcd68b7a17394ee2e89cf025d7f422ae5d06898c56ace6a39267c94  
fa8a4c226ce5a115a27e283da2b885487af866f4a1a3d74703d1f1c56f278837  
8fe5aed8da79dc420a03a84302e63e19601f122bcded20b656e60f1063487298c
```

Now, let's start those containers:

```
GAD-15-Pro:Docker gad$ docker start cEOS-1  
cEOS-1  
GAD-15-Pro:Docker gad$ docker start cEOS-2  
cEOS-2
```

Let's connect to them and set the hostnames. Here's cEOS-1:

```
GAD-15-Pro:Docker gad$ docker exec -it cEOS-1 Cli  
localhost>en  
localhost#conf  
localhost(config)#hostname cEOS-1  
cEOS-1(config)#
```

Here's cEOS-2:

```
GAD-15-Pro:~ gad$ docker exec -it cEOS-2 Cli  
localhost>en  
localhost#conf  
localhost(config)#hostname cEOS-2  
cEOS-2(config)#
```

If, like me, the first thing you would do is a `show lldp neighbor`, you're in for disappointment because the Linux bridges that we're using for default networking consume L2 link-local multicast frames.

We can, however, prove connectivity by using a simple IP setup.

First, I add an IP address to cEOS-1's Ethernet 1 interface:

```
cEOS-1(config)#int e1
cEOS-1(config-if-Et1)#no switchport
cEOS-1(config-if-Et1)#ip address 10.0.0.1/24
```

Next, I configure a matching IP on cEOS's Ethernet 1 interface:

```
cEOS-2(config)#int e1
cEOS-2(config-if-Et1)#no switchport
cEOS-2(config-if-Et1)#ip address 10.0.0.2/24
```

Can we ping? Yes, we can!

```
cEOS-2(config-if-Et1)#ping 10.0.0.1
PING 10.0.0.1 (10.0.0.1) 72(100) bytes of data.
80 bytes from 10.0.0.1: icmp_seq=1 ttl=64 time=23.8 ms
80 bytes from 10.0.0.1: icmp_seq=2 ttl=64 time=19.1 ms
80 bytes from 10.0.0.1: icmp_seq=3 ttl=64 time=20.3 ms
80 bytes from 10.0.0.1: icmp_seq=4 ttl=64 time=7.27 ms
80 bytes from 10.0.0.1: icmp_seq=5 ttl=64 time=11.6 ms

--- 10.0.0.1 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 67ms
rtt min/avg/max/mdev = 7.275/16.447/23.878/6.080 ms, pipe 3, ipg/ewma
16.803/19.793 ms
cEOS-2(config-if-Et1)#

```

I've set up these two cEOS containers in the same way in which I might have set up two vEOS VMs, but that's not really what cEOS is usually used for. cEOS is desirable because it can access the merchant silicon in a white-box switch. That's not something that the average user sitting at home would likely need to do, but knowing how cEOS can be installed is a valuable skill and can even allow you to run software features such as eAPI without the resource overhead required

by a full VM running vEOS.

Some Things to Watch Out For

As we've seen, Docker using Linux bridges means that Layer 2 (L2) protocols like Link Layer Discovery Protocol (LLDP) don't seem to work properly. To get around this limitation, you can build something called *veth pairs*, but that is beyond what I wanted to cover here. You can also force Linux to forward LLDP protocol data units (LLDPPDUs), which is beyond the scope of this book.

If you have more than three interfaces, there is no guarantee as to the order of the networks assigned to the Ethernet interfaces. We can work around this by building a custom Docker binary, but that's way outside the scope of this book.

For details on building a lab with cEOS, check out some of the excellent articles available on [Arista's EOS Central](#).

Containers in EOS

In modern versions of EOS, Docker is included in the OS, as evidenced by the fact that I can issue the `docker` command from Bash just like I did on my Mac in the previous section. This example is run on EOS version 4.21.1F:

```
[admin@Arista ~]$ docker
Usage: docker COMMAND
A self-sufficient runtime for containers
```

Options:

--config string	Location of client config files (default "/home/admin/.docker")
-D, --debug	Enable debug mode
--help	Print usage
-H, --host list	Daemon socket(s) to connect to
-l, --log-level string	Set the logging level ("debug" "info" "warn" "error" "fatal") (default "info")
--tls	Use TLS; implied by --tlsverify
--tlscacert string	Trust certs signed only by this CA (default "/home/admin/.docker/ca.pem")
--tlscert string	Path to TLS certificate file (default "/home/admin/.docker/cert.pem")
--tlskey string	Path to TLS key file (default "/home/admin/.docker/key.pem")
--tlsverify	Use TLS and verify the remote
-v, --version	Print version information and quit
[-- output truncated --]	

Just to show off a bit, this works on the small Arista 7010T, and on my internet-connected example, I can even run Hello-World like I did on my Mac. To do so, you must run **sudo** in EOS:

```
[GAD@Arista-7010T ~]$ sudo docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
d8aec4eeb95f: Pull complete
Digest: sha256:2557e3c07ed1e38f26e389462d03ed943586f744621577a99efb773
24b0fe535
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working
correctly.
```

To generate this message, Docker took the following steps:

1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub. (i386)
3. The Docker daemon created a new container from that image which runs the executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

To try something more ambitious, you can run an Ubuntu container with: `$ docker run -it ubuntu bash`

Share images, automate workflows, and more with a free Docker ID: <https://hub.docker.com/>

For more examples and ideas, visit: <https://docs.docker.com/get-started/>

Really, though, to allow EOS to properly manage containers, you should do the configuration through EOS, which has a set of commands included to make that possible. This allows the existing workflow to be maintained given that most people manage their switches using the CLI (or applications like CloudVision that manage the CLI), so it makes sense to keep things consistent.

Let's install Ubuntu on an Arista switch, this time on an internet-connected 7050TX-96:

```
Arista#sho ver
Arista DCS-7050TX-96-F
Hardware version: 11.00
Serial number: JPE15101403
System MAC address: 001c.739c.e995

Software image version: 4.21.1F
Architecture: i386
Internal build version: 4.21.1F-9887494.4211F
Internal build ID: 1497e24b-a79b-48e7-a876-43061e109b92

Uptime: 0 weeks, 0 days, 0 hours and 20 minutes
Total memory: 3817916 kB
Free memory: 2725992 kB
```

First, we need to get an image just like we did with the Docker command line, but we're going to do it in the CLI by using the `container-manager pull image-name` command:

```
Arista#container-manager pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu
Digest: sha256:7a47ccc3bbe8a451b500d2b53104868b46d60ee8f586077c650210
Status: Image is up to date for ubuntu:latest
```

There are two aspects of running containers on EOS that we need to manage: the Docker daemon and the containers. To do either, we need to get into container manager configuration mode:

```
Arista(config)#container-manager
Arista(config-container-mgr)#
```

We're going to focus on getting our container running first, so we need to name a container. I'm calling it Ubuntu with a capital "U" so that it can be differentiated from the image name that has a lowercase "u":

```
Arista(config-container-mgr)#container Ubuntu
```

Within the container configuration mode, we then need to specify the image to use:

```
Arista(config-container-mgr-container-Ubuntu)#image ubuntu
```

Now let's exit configuration mode (though we don't need to) and start the container:

```
Arista#container-manager start Ubuntu
Ubuntu
Arista#
```

That happened instantaneously, but there doesn't seem to be any obvious repercussions of doing that. Hmm. Let's see if we can view some status by using the `show container-manager info` command:

```
Arista#show container-manager info
Total Number of Containers: 1
Total Number of paused Containers: 0
Total Number of stopped Containers: 1
Total Number of Images: 1
Storage Driver: overlay
    Backing Filesystem: tmpfs
Logging Driver: json-file
Cgroup Driver: cgroupfs
Plugins:
    Volume: local
    Network: bridge, host, macvlan, null, overlay
ID: HWIR:PZ3I:PACW:NKAB:HSSM:ELVV:QPJT:FXCN:LWYZ:2TL3:SMKJ:BT4T
ContainerMgr Root Dir: /var/lib/docker
CPUs: 4
Total Memory: 3.64 GB
```

Interesting. Let's try `show container-manager containers`:

```
Arista#show container-manager containers
Container Name: Ubuntu
Container Id: 5a2c95b48441243a8acff9e668c05f59dd669254c825284ed191b7c
Image Name: ubuntu
Image Id: sha256:df55f9ba60336df3d942cf9a2d8c8d64e5e490e2f5c1d4ff0df0
Onboot: False
Command: /bin/bash
Created: 9 minutes ago
Ports:
State: exited
```

Hmm. Why does it say the State is *exited*? Because the container did what it was told to do and finished. What was it told to do? Run `/bin/bash`. It ran that process and exited. You see, Docker isn't really about creating a VM: it's about building a simple isolated environment to perform a task. If we'd configured our container to run Apache with a website, that website would not be running, because the command would be to run a daemon and it would exit after running that daemon. Still, I went through all of this trouble to get the Ubuntu image on my switch and I want to interact with it! Bash to the rescue:

```
Arista#bash docker run -it ubuntu
root@cd076f2ee645:/#
```

Woohoo! I'm now in a Bash shell in Ubuntu that I installed via containers through EOS, which is running on Fedora Core. Need more proof?

```
root@cd076f2ee645:/# more /etc/lsb-release
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=18.04
DISTRIB_CODENAME=bionic
DISTRIB_DESCRIPTION="Ubuntu 18.04.1 LTS"
```

In another window I want to see the status of this environment, so I use the command `show container-manager containers` in EOS:

```
Arista#sho container-manager containers
Container Name: Ubuntu
Container Id: 6711c8cce2f3b87a975f8f00c363db244471f0a75b9572cd00548b47
Image Name: ubuntu
Image Id: sha256:df55f9ba60336df3d942cf9a2d8c8d64e5e499bc515c1d4ff0df0
Onboot: False
Command: /bin/bash
Created: 38 minutes ago
Ports:
State: exited

Container Name: inspiring_goldberg
Container Id: cd076f2ee645c715905598c6a01187173f62392cef34d7f28cfdd745
Image Name: ubuntu
Image Id: sha256:df55f9ba60336df3d942cf9a2d8c8d64e5e499bc515c1d4ff0df0
Onboot: False
Command: /bin/bash
Created: 2 minutes ago
Ports:
State: running
```

Why are there two? The first one is the Ubuntu container that I created and started through EOS that has the state of `exited`. The second one, which has the odd name of `inspiring_goldberg`, with a state of

`running`, is the container in which I'm running my interactive Bash session in the other window. When I ran the `docker run -it ubuntu` command, the `-it ubuntu` portion specified the *image* that we pulled, and not the *container* that we made. That's an important distinction. An image is something you either download from a repository or make yourself, whereas a container is an environment that runs an image. Think of an image like a disk-image and a container like a VM, and you can kind of see the distinction.

When I issued the `docker run -it ubuntu` command, Docker created a new container on the fly and made up a name for it, which was `inspiring_goldberg`. Let's see what happens if we exit that container and fire up another one using the same command:

```
root@cd076f2ee645:/# exit
exit
Arista#bash docker run -it ubuntu
root@f900f92635d1:/#
```

If you look closely, you can see that the hostname has changed from `cd076f2ee645` to `f900f92635d1` because this is an entirely new “system” as far as Ubuntu is concerned. What's not obvious reading this is that killing and starting the new container happened instantaneously. If these were VMs, it would have taken a while for the VM to boot. There is no real booting with containers, which is one of their most powerful features. Let's see what the status looks like in my other window now:

```
Arista#sho container-manager containers
Container Name: Ubuntu
Container Id: 6711c8cce2f3b87a975f8f00c363db244471f0a75329f2cd00548b47
Image Name: ubuntu
```

```
Image Id: sha256:df55f9ba60336df3d942cf9a2d8c8d64e5e490e2f15c1d4ff0df0
Onboot: False
Command: /bin/bash
Created: 38 minutes ago
Ports:
State: exited
```

```
Container Name: inspiring_goldberg
Container Id: cd076f2ee645c715905598c6a01187173f62392ced1f8f28cfdd745
Image Name: ubuntu
Image Id: sha256:df55f9ba60336df3d942cf9a2d8c8d64e5e490e2f5c1d4ff0df0
Onboot: False
Command: /bin/bash
Created: 2 minutes ago
Ports:
State: exited
```

```
Container Name: hungry_jones
Container Id: f900f92635d1e402be38934d51c0bd29ef93f229ad1350e4104d2f0b
Image Name: ubuntu
Image Id: sha256:df55f9ba60336df3d942cf9a2d8c8d64e5e490bc515c1d4ff0df0
Onboot: False
Command: /bin/bash
Created: 2 minutes ago
Ports:
State: running
```

The container **inspiring_goldberg** that was running before now has a state of **exited**, and there's a new container called **hungry_jones**, which is the currently running container.

Let's more closely simulate how a container might be used in the real world. Back in EOS, I'm going to alter the configuration for the Ubuntu container. Here's the current configuration:

```
Arista#configure
Arista(config)#container-manager
Arista(config-container-mgr)#container Ubuntu
Arista(config-container-mgr-container-Ubuntu)#sho active
container-manager
  container Ubuntu
    image ubuntu
```

```
command /bin/bash
```

Let's change the command from `/bin/bash` to `/bin/sleep 300`. This spawns the sleep process that will do nothing for 300 seconds (five minutes):

```
Arista(config-container-mgr-container-Ubuntu)#command /bin/sleep 300
Arista(config-container-mgr-container-Ubuntu)#^z
Arista#
```

Next, start that container:

```
Arista#container-manager start Ubuntu
Ubuntu
```

Add the `brief` modifier to the `show container-manager containers` command:

```
Arista#sho container-manager containers brief
Container          Image      Onboot     State      Command
-----
Ubuntu            ubuntu     False      running
/bin/sleep 300
hungry_jones      ubuntu     False      exited     /bin/bash
inspiring_goldberg  ubuntu     False      exited     /bin/bash
```

After five minutes elapse, the status changes:

```
Arista#sho container-manager containers brief
Container          Image      Onboot     State      Command
-----
Ubuntu            ubuntu     False      exited
/bin/sleep 300
hungry_jones      ubuntu     False      exited     /bin/bash
inspiring_goldberg  ubuntu     False      exited     /bin/bash
```

Want to have your container automatically spin up when you boot the switch? Add the `on-boot` command to the container:

```
Arista(config-container-mgr-container-Ubuntu)#on-boot
```

What happens to my images when I reload? After doing a `write mem` and a reload, I log in and show my containers:

```
Arista#sho container-manager containers brief
  Container          Image      Onboot     State      Command
  -----  -----  -----  -----  -----
  Ubuntu           ubuntu     True      running   /bin/sleep 300
```

Let's think about what happened here, because it's important to understand how and why this worked. Remember, the filesystem in EOS is destroyed when the switch reboots except for anything on *flash*: and *drive*:. The Docker containers don't live there, so they were destroyed, too. So how did this work?

This switch is connected to the internet so that when we rebooted and the container-manager commands took effect, Docker went to the Docker Hub on the internet, retrieved the `ubuntu` image from there, downloaded it, and installed it in order to start the container. That might or might not be very cool depending on your needs and requirements. The good news is that you can change this behavior by *backing up* a container locally:

```
Arista#container-manager backup container Ubuntu
Container Ubuntu has been committed. Backing up created
sha256:0a6daefa5cc269f1
53871fe1ae1f2e8c95976b8348830e53432ef44b717e4eea image at
/mnt/flash/.containermgr/Ubuntu.tar
```

Sure enough, looking in the hidden directory `.containermgr` on `/mnt/flash`, we see the backup of our container:

```
Arista#bash ls -al /mnt/flash/.containermgr
```

```
total 87916
drwxrwx--- 2 root eosadmin 4096 Feb  7 15:06 .
drwxrwx--- 8 root eosadmin 4096 Feb  7 15:06 ..
-rwxrwx--- 1 root eosadmin 90016256 Feb  7 15:06 Ubuntu.tar
```

Be careful with this because that is a 90 MB file. If you have many containers and you back them all up, you can consume a fair bit of your flash drive, which is typically not very large.

Now, when the switch reboots, instead of going to the repository for the container's relevant image, it just grabs it from the local resource.

To view what containers have been backed up, use the `show container-manager backup` command:

```
Arista#sho container-manager backup
      Files          Directory
-----
Ubuntu          /mnt/flash/
```

To remove one, use the `container-manager backup remove container-name` command:

```
Arista#container-manager backup remove Ubuntu
Arista#sho container-manager backup
      Files          Directory
-----

```

Conclusion

Containers are changing the world in a similar way that VMs did, and Arista is again on the front lines with support in EOS. With the ability to run containers within EOS, coupled with the ability for EOS to be run in a container (cEOS), Arista has all the bases covered.

How can you use containers to your benefit? That's a tough question to answer because every environment is different, but knowing that containers exist and the ways that they are supported in EOS might just give you the cutting edge you need to solve a problem in a unique and powerful way, that's the kind of thing that still excites me about Arista to this day.

Chapter 32. vEOS

When a software system is designed from the ground up to be flexible, scalable, and extensible, wonderful benefits occur. One of the marvelous things about EOS is that it can run without a physical switch, and with vEOS, we can build a functioning lab on our own laptops. We could also build labs with VMware or Fusion or Parallels or Virtual Box. In fact, I'm going to show you how to do just that, but first, let's talk about what vEOS is and, perhaps just as importantly, what it is not.

vEOS is EOS, but built as a package for use as a virtual machine (VM). Though it would be wonderful for vEOS to support every feature that EOS can deliver, that is currently not possible, and the reason for that is simple: many of EOS's features are dependent on the Application-Specific Integrated Circuits (ASICs) used in the switches. That's why a feature like *tap aggregation* is available only on certain platforms. Although it is technically feasible to write emulators that would give the appearance of the same functionality, that's not what vEOS is about. Remember, vEOS is the same code as EOS—it's just compiled for a VM environment. Remember, too, that Arista prides itself on the idea that all devices in the Arista lineup can use the same binary, so if vEOS were to be a separate code image, disparities would begin to surface, and that's not how Arista likes to operate.

In reality, the limitations center around hardware, so think of it this

way: If you had an Arista 7150 switch, you wouldn't be able to configure Direct Flow because it's not supported on that switch's ASIC. So it is with vEOS. Don't despair, though, because there are so many features that do work in vEOS that you'll likely not even care about the limitations. Any software-based feature will work, so features like Link Layer Discovery Protocol (LLDP), Multichassis Link Aggregation (MLAG), Zero-Touch Provisioning (ZTP), Spanning Tree, Bash, tcpdump, Python, Extensions, VMTracer, email, Event Manager, Scheduler, routing, Event Monitor, and so on, all work in vEOS. That makes vEOS a great tool for testing control-plane functionality and interoperability.

Additionally, as of 2019, there are two versions of vEOS, which are vEOS-Lab and vEOS-Router. Here are the differences:

vEOS-Lab

vEOS-Lab is designed to be used as a simulation tool in a lab environment. vEOS-Lab is free to use, and all you need to do to get it is have a login to the Arista.com website. Even as a simple guest account, you can now download vEOS!

vEOS-Router

vEOS-Router is designed for use in cloud services such as Amazon Web Services (AWS), Google Cloud platform (GCP), and Microsoft Azure. The idea behind vEOS-Router is that you can use an Arista-supported EOS device to connect your network to the cloud instead of some generic virtual router solution from the cloud provider. vEOS-Router has a couple of advantages over vEOS-Lab, the most important being a vastly improved packet forwarding capability and the ability to support Virtual Private Network (VPN) and IP Security (IPSec). vEOS-Router is also a licensed product, so you need to pay for its use. If you have Azure or AWS, you can

actually install and pay for vEOS-Router directly through those services' marketplaces.

One of the really cool things that Arista has done in the past few years is to combine the Aboot and vEOS images into a single combined image that makes building VMs a whole lot simpler. Back in 2012, I wrote a blog post called "Building a Virtual Lab with Arista vEOS and VirtualBox," which is one of the most popular blog posts on my site, but it's a bit outdated at this point, so I'm going to write a completely new updated version here.

VEOS in VirtualBox

Let's build a real usable lab in VirtualBox. First, you need VirtualBox if you don't already have it. It's free, and you can get it from [the VirtualBox website](#). I'm using a Mac, but the company has images for Windows, Linux, and Solaris, too. With VirtualBox installed, the next thing we need is a vEOS image. You need to have a login to the Arista web page to download this, and it's available under Support on the Software Downloads page. I've gone ahead and downloaded both and installed VirtualBox on my Mac, so you're on your own for those two steps. From here on in I'll assume that you've done both, as well. To whet your appetite, [Figure 32-1](#) shows the lab that we're going to build.

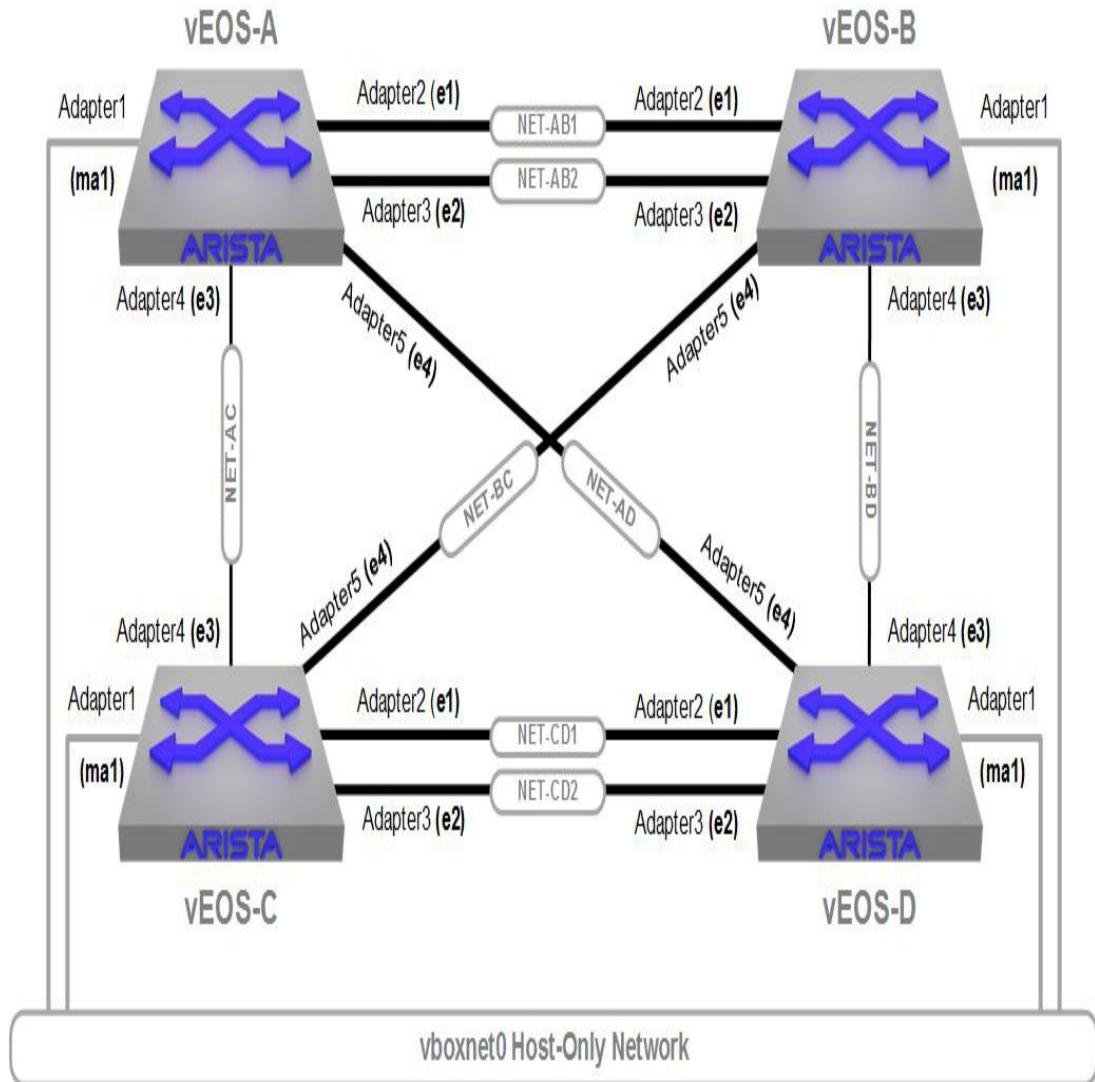


Figure 32-1. Our super-cool vEOS lab

After you've installed VirtualBox, fire it up and click the “new” button. A dialog box opens in which you must name your new VM. I'm going to create a base example called vEOS-Base that we'll replicate in later steps. Make sure you choose Linux and Fedora 64-bit for the Type and Version, as shown in Figure 32-2, and then click Continue.

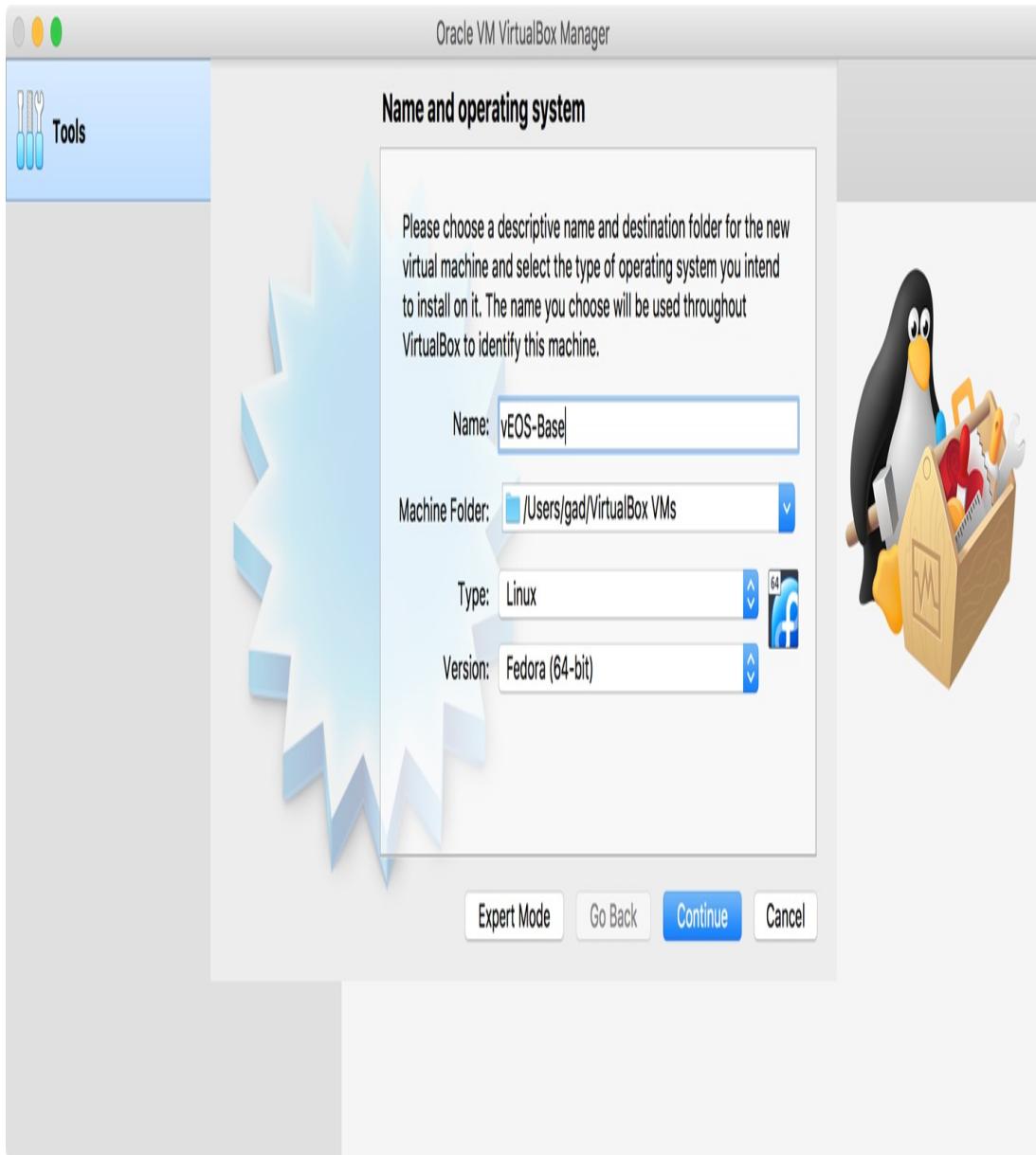


Figure 32-2. The new VM dialog box

Creating the Base VM

Next, select the amount of memory for the VM. More is better, of course, but because I'm going to create a few and my Mac has only 16 GB of RAM, I'm going to stick with 2048 MB, as shown in Figure 32-3.

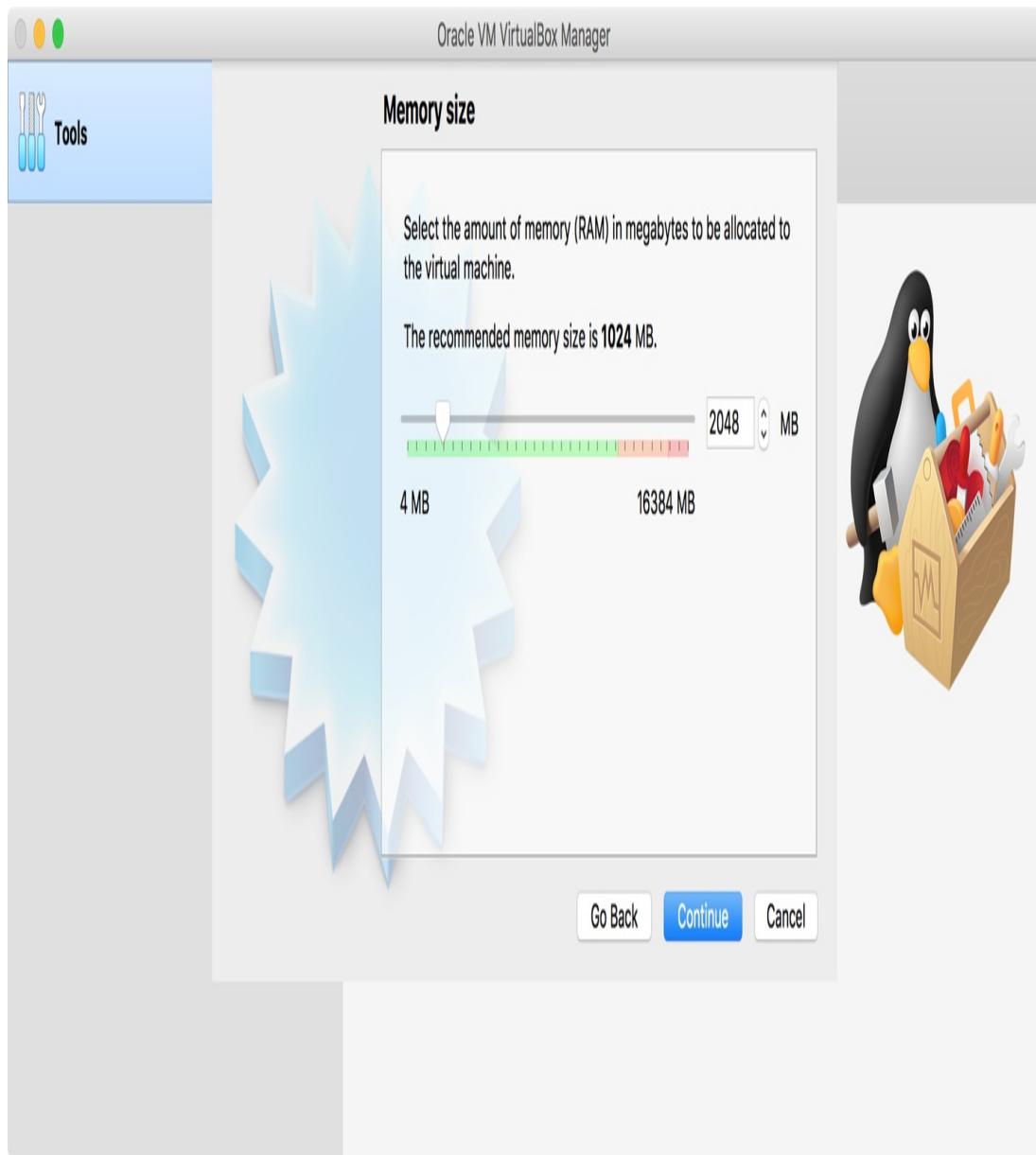


Figure 32-3. Selecting the memory size

Click Continue. You then have the choice of how to deal with the hard disk. The default is “Create a virtual hard disk now,” but for this exercise, choose “Use an existing virtual hard disk file,” and then click the small folder icon to the right of the pulldown menu labeled Empty, as shown in Figure 32-4.

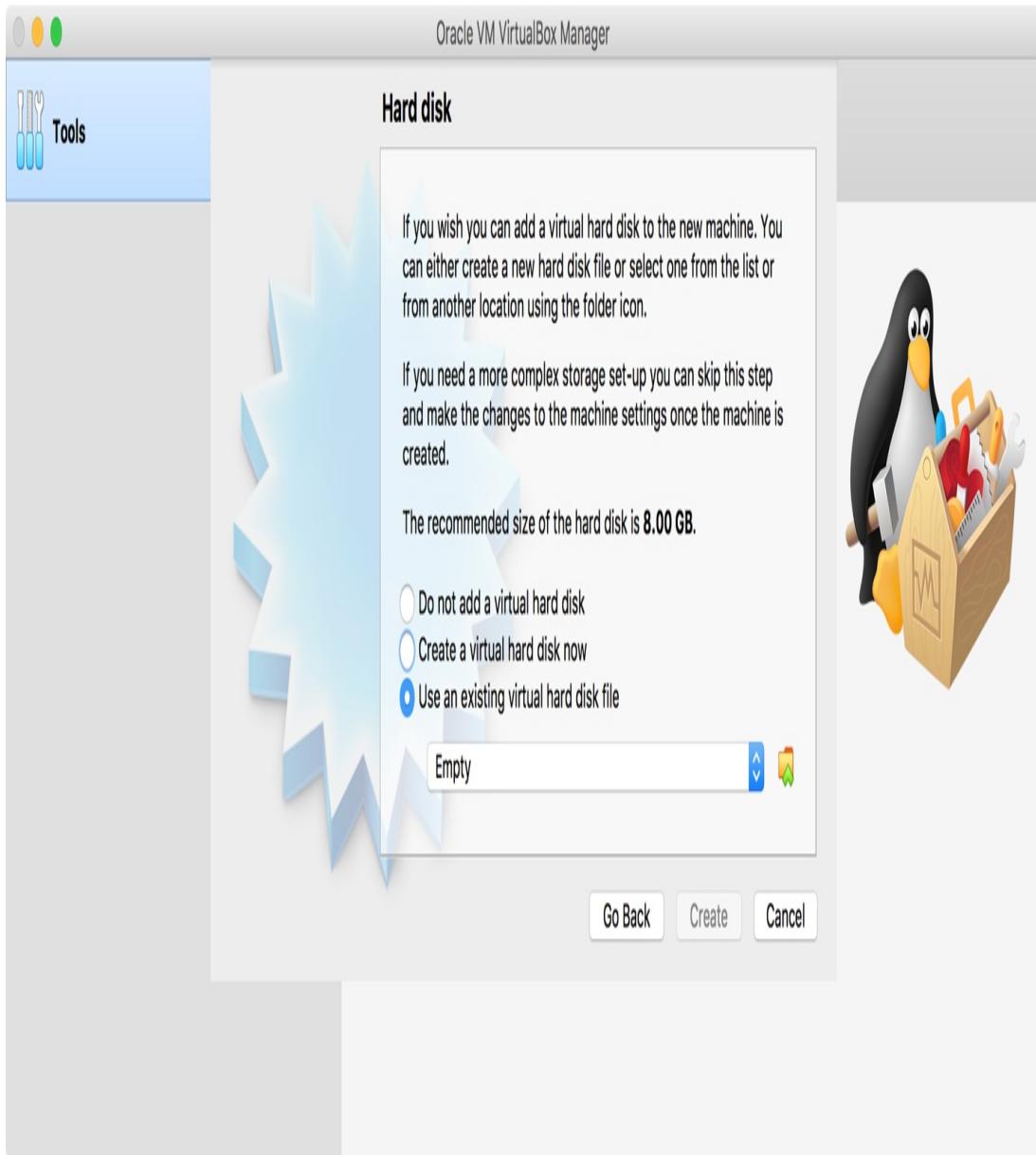


Figure 32-4. The hard drive dialog box

In the dialog box that opens, in the upper-left corner of the center pane, click the Add icon, as depicted in Figure 32-5. Another dialog box opens in which you can choose a virtual hard disk file—this is the *vEOS-lab-4.21.1.1F.vmdk* file that I downloaded earlier. Because I didn't do anything with the file, it's in my Downloads folder, so I point there, select the file, and click Open. See Figure 32-5 because a picture is worth at least 137 words.

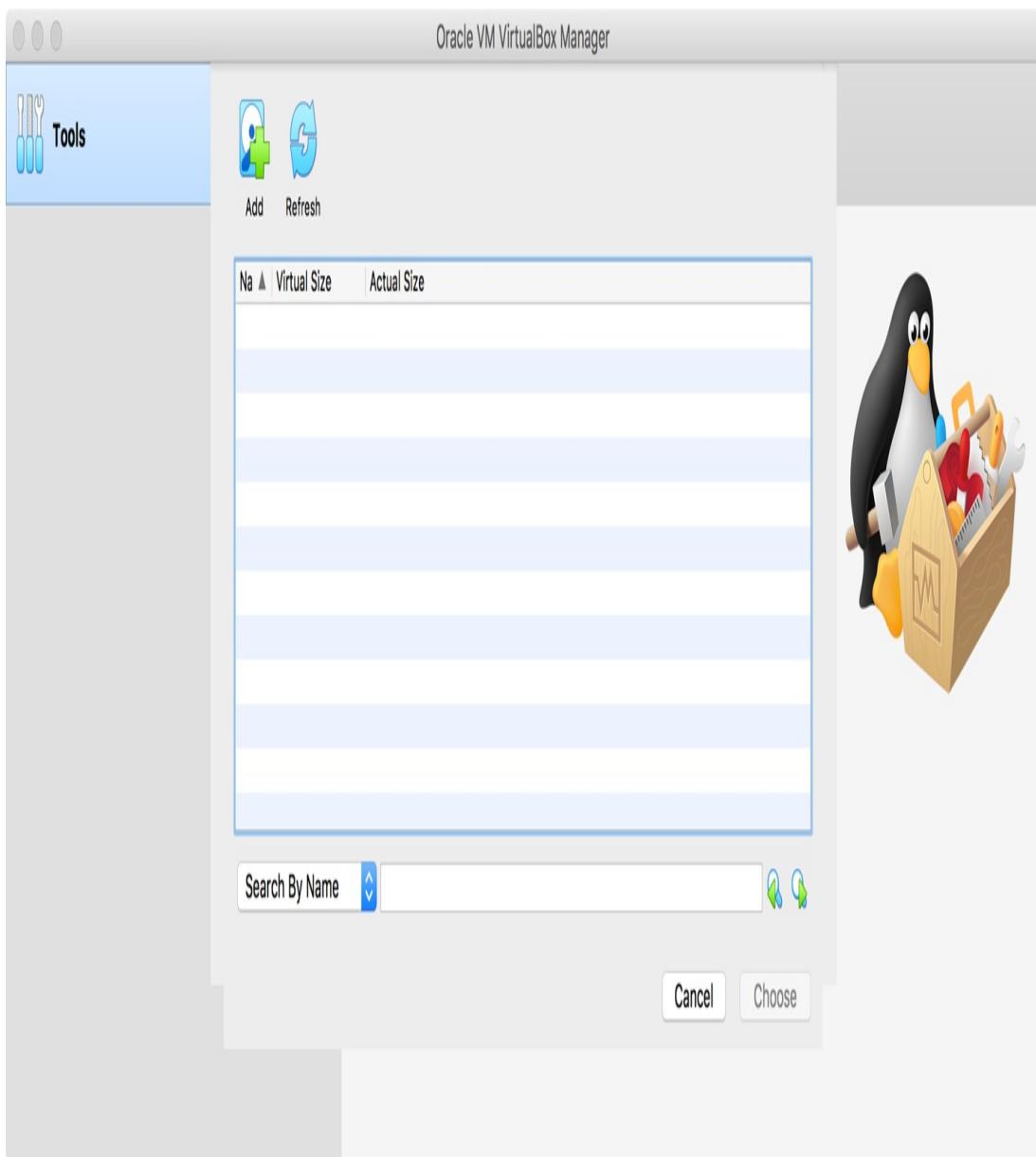


Figure 32-5. Selecting a VMDK files

With the vEOS file selected and opened, it should now appear in the list of disk images. Make sure it's selected and then click Choose. You should return to the Hard Disk dialog, which should now show the chosen VMDK file, as shown in Figure 32-6.

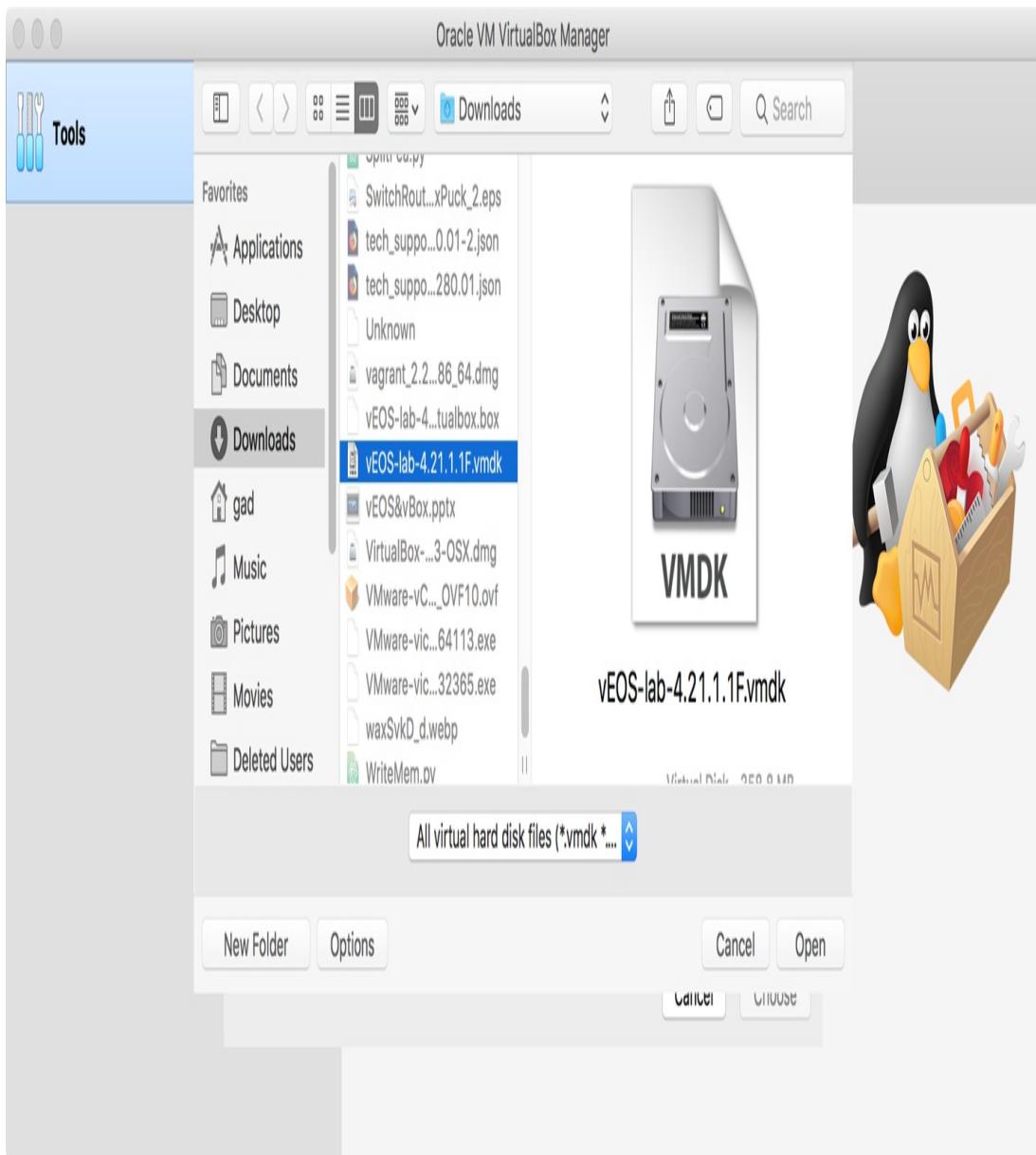


Figure 32-6. Back to the hard disk dialog box

Click the Create button. Your VM should now have been created, and you should see a screen similar to Figure 32-7. Congrats! Only 934 screenshots to go!

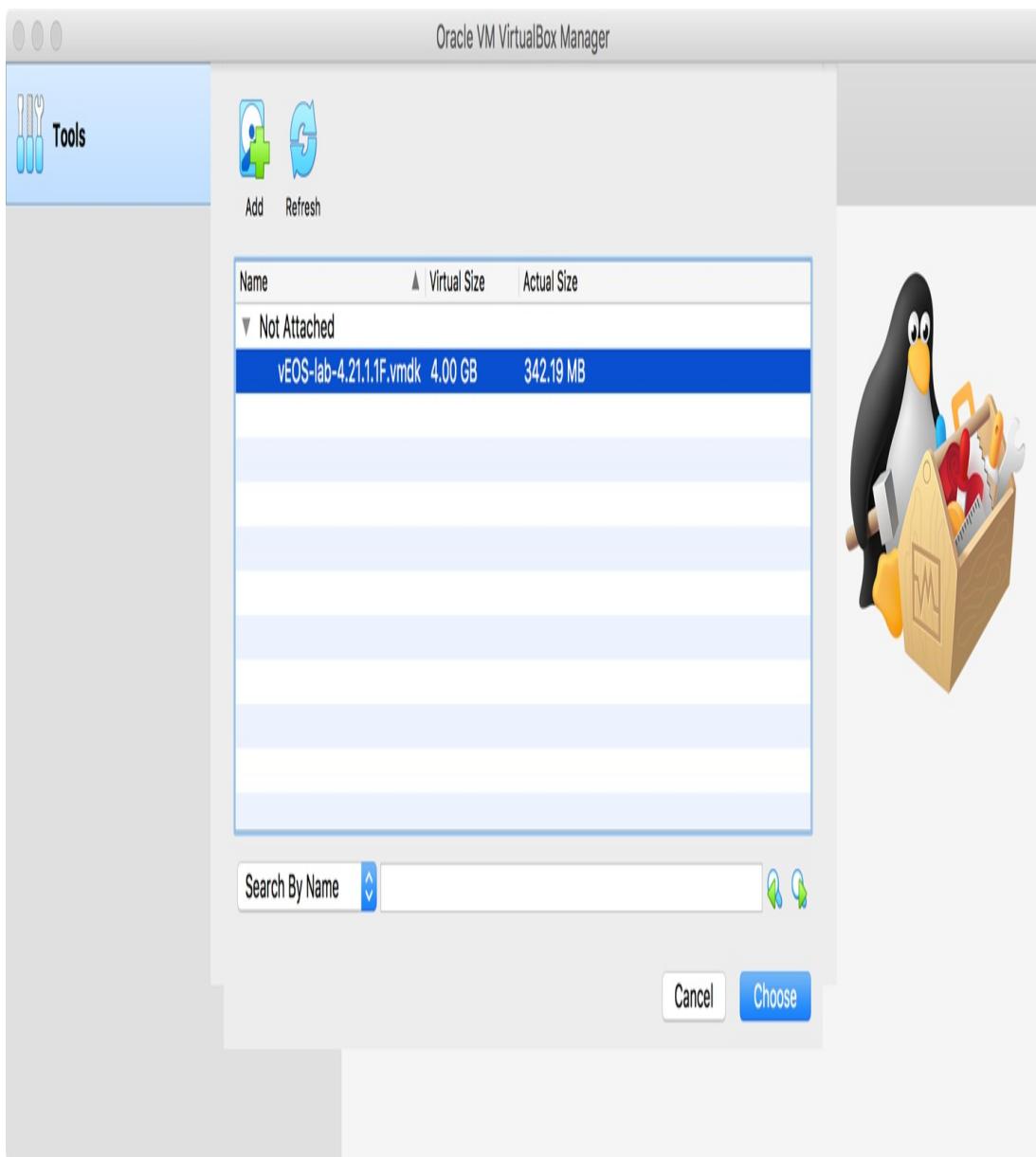


Figure 32-7. Base vEOS image created

Before we continue, we must create a host network using the Host Network Manager. From the main pull-down menu, click File, and then choose Host Network Manager. If you've never done this before, you'll have an empty page. Click the Create button, which gives you a default network named "vboxnet0." I've changed the IP address on mine from the default 192.168.x.x range to 10.0.0.0, but anything will work so long as you're consistent. To change the IP address, right-click

the “vboxnet0” line and choose “properties,” which opens a dialog box similar to that shown in Figure 32-8.

This is the network that will allow the VMs to communicate with the host, which, in my case, is my Macbook Pro.



Figure 32-8. Creating a host network

When you’re done, click Close. With the host network created, we can move on to the management interface on our VM. VirtualBox supports up to eight interfaces, even though it shows only four; I show you how to add additional network interfaces later on. The remaining interfaces are configured based on how they’re connected, so we’re going to hold off on those for now and just create the management interface. Back on the main VirtualBox screen, click the Settings icon (the yellow gear), and then choose Network. You then see four interfaces, which is the default number supported.

The interfaces are assigned in order in vEOS with Management-1 being the first. Thus, Adapter 2 resolves to Ethernet-1, Adapter 3 to Ethernet-2, and so on. Choose Adapter 1 and set the “Attached to” pull-down menu to “Host-only Adapter.” The name should default to “vboxnet0” now that we’ve created it. Click the “advanced” switch and then change the Adapter Type to “PCnet-FAST III” and Promiscuous Mode to “Allow VMs.” The options should look similar to the one depicted in Figure 32-9.

vEOS-Base - Network

General System Display Storage Audio Network Ports Shared Folders User Interface

Adapter 1 Adapter 2 Adapter 3 Adapter 4

Enable Network Adapter

Attached to: Host-only Adapter

Name: vboxnet0

Advanced

Adapter Type: PCnet-FAST III (Am79C973)

Promiscuous Mode: Allow VMs

MAC Address: 080027782FAF

Cable Connected

Port Forwarding

Cancel OK

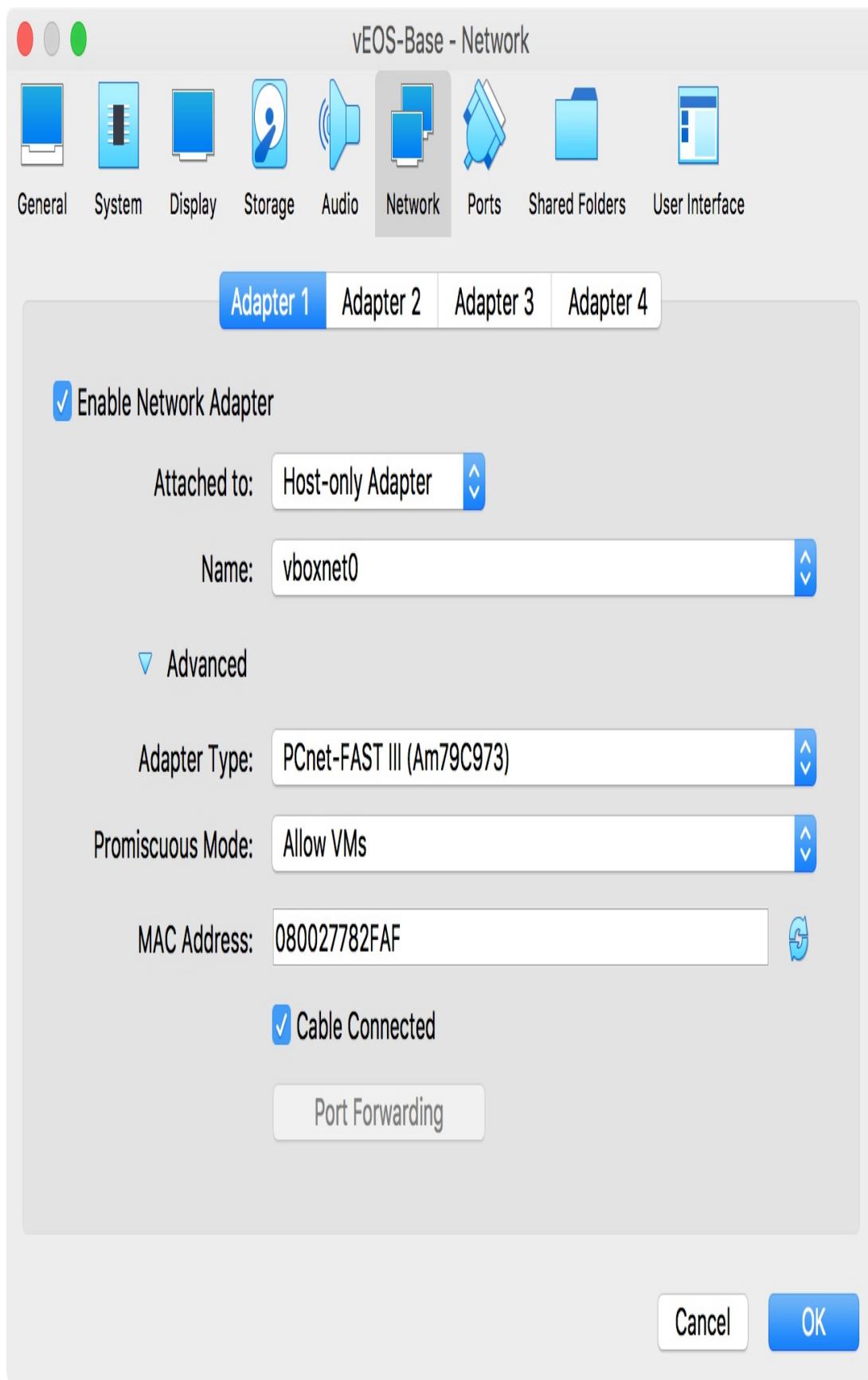
A screenshot of a Mac OS X-style configuration window titled "vEOS-Base - Network". The window has a toolbar with icons for General, System, Display, Storage, Audio, Network (selected), Ports, Shared Folders, and User Interface. Below the toolbar is a tab bar with "Adapter 1" (selected), "Adapter 2", "Adapter 3", and "Adapter 4". The main content area shows the configuration for Adapter 1. It includes a checked checkbox for "Enable Network Adapter", dropdown menus for "Attached to" (set to "Host-only Adapter") and "Name" (set to "vboxnet0"), and a collapsed "Advanced" section. The "Advanced" section contains dropdown menus for "Adapter Type" (set to "PCnet-FAST III (Am79C973)") and "Promiscuous Mode" (set to "Allow VMs"), and a text input field for "MAC Address" (set to "080027782FAF"). There is also a checked checkbox for "Cable Connected" and a "Port Forwarding" button. At the bottom are "Cancel" and "OK" buttons.

Figure 32-9. Configuring the management interface

With the management interface created, let's go and set the other three to useful defaults so that we need make only minimal changes to the other VMs when we clone this one. You can configure all four interfaces before you click OK.

Go to each of the other three adapters and set them the same way, making sure that you click Enable Network Adapter for each, but for these interfaces the “Attached to” field will be “Internal Network.” This setting needs a Name, which will default to “intnet”—leave that as the default for now on all three interfaces. Again, click Advanced and set the Adapter Type to “PCnet-FAST III” and the Promiscuous Mode to Allow VMs. Each of the interface configurations should look like the example shown in [Figure 32-10](#).

vEOS-Base - Network

General System Display Storage Audio Network Ports Shared Folders User Interface

Adapter 1 Adapter 2 Adapter 3 Adapter 4

Enable Network Adapter

Attached to: Internal Network

Name: intnet

Advanced

Adapter Type: PCnet-FAST III (Am79C973)

Promiscuous Mode: Allow VMs

MAC Address: 0800279B6967

Cable Connected

Port Forwarding

Cancel OK

This screenshot shows the configuration interface for Adapter 4 in vEOS-Base. The 'Enable Network Adapter' checkbox is checked. The adapter is attached to the 'Internal Network' and is named 'intnet'. The adapter type is set to 'PCnet-FAST III (Am79C973)'. Promiscuous mode is set to 'Allow VMs'. The MAC address is '0800279B6967'. The 'Cable Connected' checkbox is checked. A 'Port Forwarding' button is visible. Buttons for 'Cancel' and 'OK' are at the bottom right.

Figure 32-10. Configuring additional interfaces

When you're done, click OK. At this point, it's a good idea to start the VM to verify that everything looks good. From the main VirtualBox window, ensure that the vEOS-Base VM is selected and then, at the top of the page, click the big green Start arrow. You might see some VirtualBox messages in the new console that pops up, but you can just close those for now. The resulting page should look like any other switch console. Note that on my Mac, the console window is too tiny to read, probably due to the Retina display and the way I have the screen set. If that's the case, go to the VirtualBox pull-down menu, click View, Virtual Screen 1, and then scale to your desired size.

The VMs can take a little longer to boot than a normal switch in my experience, and you might see some entries on the screen that you wouldn't see on a switch, but be patient and the VM should boot and look like what is shown in [Figure 32-11](#).

```
vEOS-Base [Running]
[ 0.898055] Running dosfsck on: /mnt/flash
fsck.fat 3.0.23 (2013-10-15) Copyright (C) 1994-2011 H. Peter Anvin et al
/dev/sda2: 22 files, 172138/1046527 clusters
Switching rootfs...ready.
RTNETLINK answers: No such process

Welcome to Arista Networks EOS 4.21.1.1F
method return sender=:1.0 -> dest=:1.1 reply_serial=2
New seat seat0.valid argument
Starting ConnMgr: Starting TimeAgent: [ OK ]
[ OK ]
Starting ProcMgr: [ OK ]
Starting EOS initialization stage 1: ell [ OK ]
Starting NorCal initialization: [ OK ]
Starting EOS initialization stage 2: [ OK ]
Starting Power On Self Test (POST): [ OK ]
Completing EOS initialization (press ESC to skip): _
```

Figure 32-11. vEOS booting

The VM will likely boot into ZTP mode, so follow the steps you learned in [Chapter 13](#) to disable that (unless you’re using the VMs to learn about ZTP, in which case, carry on). After you disable ZTP (remember that the switch will reboot), log in as admin with no password, and then `enable` and do a `show interface status` to confirm that your interfaces are all there. If you don’t see what’s shown in [Figure 32-12](#), you’ll need to shut down the VM and check your interface configurations in VirtualBox.

localhost login: admin
Last login: Mon Feb 11 21:13:18 on tty1
localhost>en
localhost#sho int status

Port	Name	Status	Vlan	Duplex	Speed	Type	Flags	Enca
psulation								
Et1		connected	1	full	unconf	EbraTestPhyPort		
Et2		connected	1	full	unconf	EbraTestPhyPort		
Et3		connected	1	full	unconf	EbraTestPhyPort		
Ma1		connected	routed	a-full	a-100M	10/100/1000		

localhost#



Figure 32-12. Interfaces in the vEOS-Base VM

At this point you can shut down the VM. In the VirtualBox main screen, in the left pane, right-click the VM (it should say vEOS-Base Running), choose Close, and then click “save state.” Now the real fun can begin.

Making a Real Lab Through Cloning

With the base VM created, we can now clone it into four other VMs that we’ll call vEOS-A, vEOS-B, vEOS-C, and vEOS-D. To do so, in the left pane of the main VirtualBox window, right-click the VM, and then, in the context menu that opens, choose “clone.” You are presented with a new dialog box asking about how the clone should be made. These options are very important, and your VMs won’t work properly if you don’t choose wisely. The name of the first clone will be vEOS-A. Leave the path alone, but ensure that the MAC Address Policy is set to “Generate new MAC addresses for all network adapters,” as shown in [Figure 32-13](#).

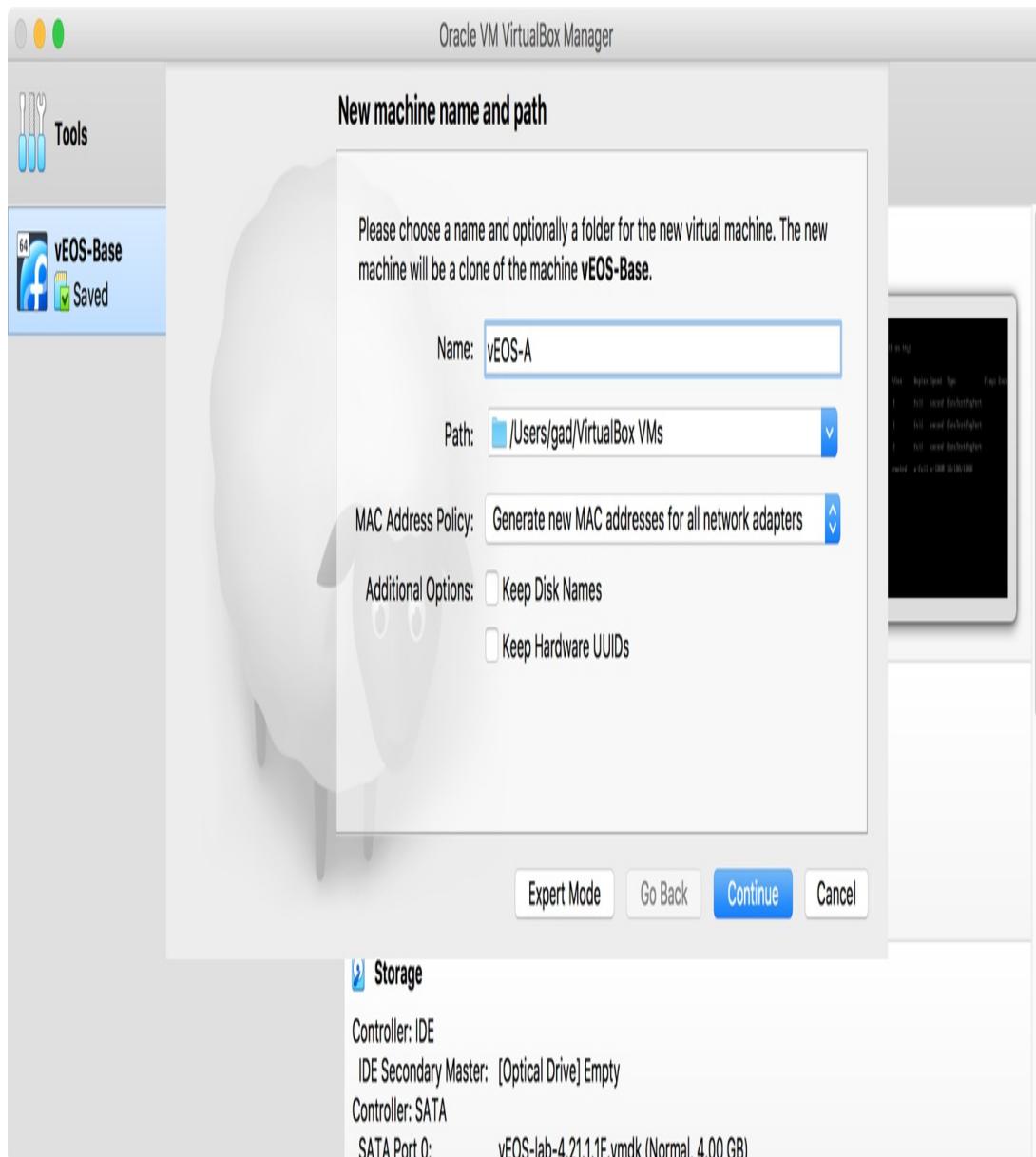


Figure 32-13. Cloning a VM

Click Continue. Another dialog box opens, in which you must pick Full Clone for the lab to work properly. Click the Clone button. You should now have a new VM in the list called vEOS-A. Follow the steps three more times, cloning the original vEOS-Base image to create VMs named vEOS-B, vEOS-C, and vEOS-D. Take a second to appreciate the fact that a sheep shows up during the cloning process, which is an homage to Dolly the sheep, who was the first living mammal ever

cloned. At the end of the cloning process you should have four VMs created and the main VirtualBox, as shown in [Figure 32-14](#).



Figure 32-14. Four cloned VMs

Do yourself a favor at this point and start each VM and change the hostname in vEOS so that they don't all say "localhost." Also, be thankful that we cloned VMs so that you don't need to disable ZTP in all three VMs. After you change the hostnames, do a write in each of

the VMs and then go back to the VirtualBox screen and right-click Close, Save State on all three of the VMs because now it's time to build the inter-VM networks.

Building the Interswitch Networks

I find this to be the most complicated part of the process, mostly because of the way that VirtualBox deals with the private inter-VM networks. This is a common issue when using switches in virtual environments because VMs weren't really designed to run switches—they were designed to run servers. As such, there is no real concept of a virtual cable, so we need to make little networks. To begin, I've created a spreadsheet that outlines how the VMs will connect. Remember that lab layout I showed you a few pages back? Well, it has a fair bit of interconnects that we need to define. You can see these in the spreadsheet shown in Figure 32-15.

VM	Adapter	Network Type	EOS Int Net Name	Description	IP Address
vEOS-A	Adapter1	Host-Only	ma1	vboxnet0	Management Network
	Adapter2	Internal Network	e1	NET-AB1	Link to vEOS-B (1)
	Adapter3	Internal Network	e2	NET-AB2	Link to vEOS-B (2)
	Adapter4	Internal Network	e3	NET-AC	Link to vEOS-C
	Adapter5	Internal Network	e4	NET-AD	Link to vEOS-D
vEOS-B	Adapter1	Host-Only	ma1	vboxnet0	Management Network
	Adapter2	Internal Network	e1	NET-AB1	Link to vEOS-A (1)
	Adapter3	Internal Network	e2	NET-AB2	Link to vEOS-A (2)
	Adapter4	Internal Network	e3	NET-BD	Link to vEOS-D
	Adapter4	Internal Network	e4	NET-BC	Link to vEOS-C
vEOS-C	Adapter1	Host-Only	ma1	vboxnet0	Management Network
	Adapter2	Internal Network	e1	NET-CD1	Link to vEOS-D (1)
	Adapter3	Internal Network	e2	NET-CD2	Link to vEOS-D (2)
	Adapter4	Internal Network	e3	NET-AC	Link to vEOS-A
	Adapter4	Internal Network	e4	Net-BC	Link to vEOS-B
vEOS-C	Adapter1	Host-Only	ma1	vboxnet0	Management Network
	Adapter2	Internal Network	e1	NET-CD1	Link to vEOS-C (1)
	Adapter3	Internal Network	e2	NET-CD2	Link to vEOS-C (2)
	Adapter4	Internal Network	e3	NET-BD	Link to vEOS-B
	Adapter4	Internal Network	e4	NET-AD	Link to vEOS-A
Host		Host-Only		vboxnet0	Management Network
					10.0.0.1/24

Figure 32-15. vEOS lab network layout

To make all of that work we need to go into the configuration for each VM and add settings for each of the network interfaces from Adapter2 through Adapter4 (not Adapter1 because that's the management interface). Be careful during this step because each interface must connect to the correct internal network in order to simulate the network that we want. This can be easily fixed if you get it wrong, but troubleshooting it can be a pain. I try to name the networks in relation to what they connect to so, for example, NET-BC is connecting vEOS-B and vEOS-C. [Figure 32-16](#) shows the network-to-interface pairing for vEOS-A Adapter 4.

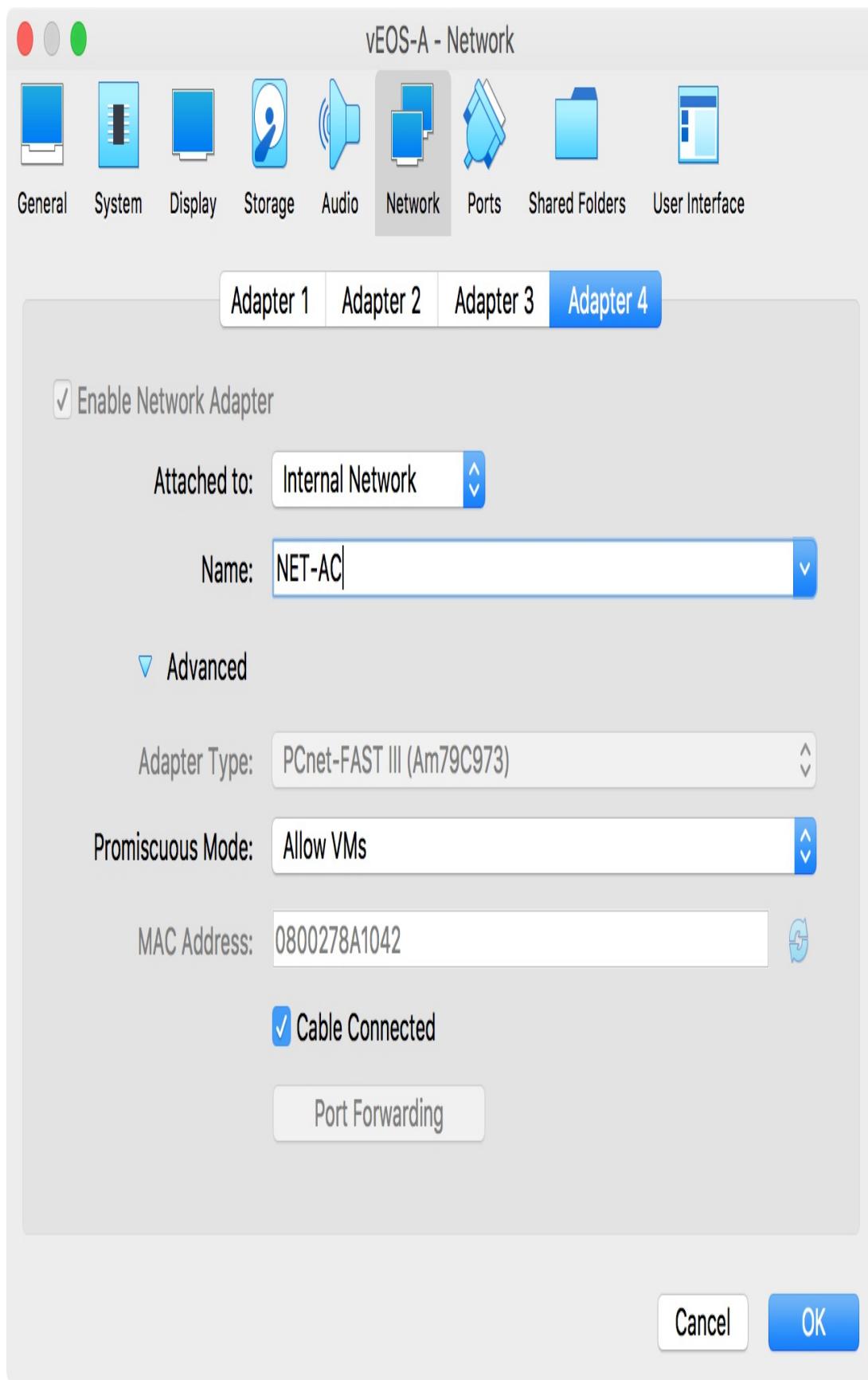


Figure 32-16. Changing Ethernet interfaces on VMs

You might have noticed that I have an Adapter5 in my spreadsheet and there's no such option in VirtualBox. Unfortunately, they wrote the GUI to support only four interfaces even though VirtualBox can support up to eight. To add another interface, we need to resort to using command-line tools. I love me some command-line tools! This is a good thing because I probably had to redo this step four times after judicious misuse of the up-arrow and enter combination burned me repeatedly.

To do these next few steps, the VMs must all be powered off. If you just suspend them, you'll get a "VM is not immutable" error, and no one wants that. From my Mac I open up the *terminal* app and type the following, which adds Adapter5, connects it to an internal network, connects to the proper network according to my spreadsheet, and then sets it to Promiscuous mode `allow-vms`.

```
GAD-15-Pro: gad$ vboxmanage modifyvm vEOS-A --nic5 intnet
GAD-15-Pro: gad$ vboxmanage modifyvm vEOS-A --intnet5 "NET-AD"
GAD-15-Pro: gad$ vboxmanage modifyvm vEOS-A -nicpromisc5 allow-
vms

GAD-15-Pro: gad$ vboxmanage modifyvm vEOS-B --nic5 intnet
GAD-15-Pro: gad$ vboxmanage modifyvm vEOS-B --intnet5 "NET-BC"
GAD-15-Pro: gad$ vboxmanage modifyvm vEOS-B -nicpromisc5 allow-
vms

GAD-15-Pro: gad$ vboxmanage modifyvm vEOS-C --nic5 intnet
GAD-15-Pro: gad$ vboxmanage modifyvm vEOS-C --intnet5 "NET-BC"
GAD-15-Pro: gad$ vboxmanage modifyvm vEOS-C -nicpromisc5 allow-
vms

GAD-15-Pro: gad$ vboxmanage modifyvm vEOS-D --nic5 intnet
GAD-15-Pro: gad$ vboxmanage modifyvm vEOS-D --intnet5 "NET-AD"
GAD-15-Pro: gad$ vboxmanage modifyvm vEOS-D -nicpromisc5 allow-
```

vMS

That might seem like a lot of technical typing, but the only thing that changes is the VM name and, in the second line of each VM’s configurations, the network name. It’s really just a matter of doing the first VM and then carefully up-arrowing and changing the few characters where needed. The “carefully” part is what caused me grief.

After we have configured the first four interfaces via the VirtualBox GUI and we add the last one via the command line, we should have a working network.

To make your life simpler (which is always a good thing), start up your VMs, log into each on the console, and then do the following:

If you haven’t already, set the hostnames:

```
localhost#conf
localhost(config)#hostname vEOS-A
```

Set the IP address on the management interface for each vEOS VM according to the values in the spreadsheet in [Figure 32-15](#):

```
vEOS-A(config)#int ma1
vEOS-A(config-if-Ma1)#ip address 10.0.0.101/24
```

Finally, configure the VMs so that you can log in as admin without a password, unless you want to enter a password every time you connect to them.

```
vEOS-A(config-if-Ma1)#aaa authentication policy local allow-
nopass
```

After you do this on all of your VMs, check your network connectivity by using the `show lldp neighbor` command in each of them:

```
vEOS-A(config)#sho lldp nei
Last table change time  : 0:01:33 ago
Number of table inserts : 10
Number of table deletes : 3
Number of table drops   : 0
Number of table age-outs: 0

Port      Neighbor Device ID          Neighbor Port ID
TTL
Et1       vEOS-B                  Ethernet1
120
Et2       vEOS-B                  Ethernet2
120
Et3       vEOS-C                  Ethernet3
120
Et4       vEOS-D                  Ethernet4
120
Ma1       vEOS-C                  Management1
120
Ma1       vEOS-B                  Management1
120
Ma1       vEOS-D                  Management1
120
vEOS-A(config)#
```

You'll likely see all of the other VMs showing up on the management interface, which is normal in this environment due to the way that the virtual bridge works.

From this point on, you can start your VMs in headless mode, which means that they will start without consoles popping up all over your screen. The consoles are great when you need to be on the console (for things like ZTP and Aboot), but they're a pain in the butt if you're used to using a real Secure Shell (SSH) client. You can now use your favorite SSH client or even just the terminal window (if you have a

Mac) to connect to your VMs:

```
GAD-15-Pro:~ gad$ ssh -l admin 10.0.0.101
Last login: Mon Feb 11 23:20:01 2019 from 10.0.0.1
vEOS-A>en
vEOS-A#
```

At this point, if you don't think you'll need to do any more cloning, you can delete the original vEOS-Base VM that we created. I like to keep it around because it's a great way to spin up new standalone VMs for testing, but I also dislike clutter, so deleting it is fine, too.

At this point, your lab should be up and you can toy with anything you might want to build. Use it to test out some MLAG knowledge, or even Equal-Cost MultiPathing (ECMP) using Border Gateway Protocol (BGP). The world is your oyster. Or something.

Automating VirtualBox Builds

There's a very cool tool out there called Vagrant that can automate this entire process. I didn't just do that because I think there's a fair bit of knowledge and experience to be gained by building it by hand. There are a bunch of cool Vagrant network builds available on GitHub that you can find by searching for *Vagrant vEOS* using your favorite internet search engine. Note that this is not a way around getting vEOS because you must have a vEOS image and VirtualBox installed for any of these to work. Note also that if you decide to play around with this capability that you'll need the vEOS image with the *virtualbox.box* filename extension and not the normal *.vmdk* file.

vEOS in an EOS VM

Did you know that you can run VMs on your Arista switch? This doesn't mean that you should, but you absolutely can by using the EOS **virtual-machine** commands. There is a fair bit of risk involved in doing this because a VM can consume resources on your switch, and one of the worst things that can happen to an Arista switch is running out of memory, so don't do this just because it's cool. Having written that, I'm absolutely going to do this because it's cool. Hell, I'm going to go crazy and take it a step further by building two VMs. Why? Because I have a kick-ass 7280R switch with 32 GB of RAM just begging to be utilized. And it's cool:

```
Arista(config)#sho ver
Arista DCS-7280SR-48C6-M-F
Hardware version: 11.03
Serial number: SSJ16462978
System MAC address: 2899.3a26.515d

Software image version: 4.21.1F
Architecture: i386
Internal build version: 4.21.1F-9887494.4211F
Internal build ID: 1497e24b-a79b-48e7-a876-43061e109b92

Uptime: 0 weeks, 5 days, 2 hours and 47 minutes
Total memory: 32458980 kB
Free memory: 28324816 kB
```

I wanted to do something that would be fun, weird, and maybe even show off some more Arista...ness. And that's why I'm going to build this crazy lab, so buckle up and hang on because it's about to get weird.

First, let's create a VM. Because I am utterly devoid of imagination after writing for nine hours straight (it's currently 3 a.m.), I'm going to call it GAD-1.

```
Arista(config)#virtual-machine GAD-1
```

Next, we need to specify the image for the VM. I'd previously procured a copy of vEOS version 4.21.1.1F, which you'll notice is a minor revision different than the version on my switch (4.21.1 versus 4.21.1.1) and which also has a file extension of *vmdk* instead of *swi*.

```
Arista(config-vm-GAD)#disk-image flash:vEOS-lab-4.21.1.1F.vmdk
```

Because memory is a valuable commodity in EOS, the default memory size that a VM will be allocated is a paltry 128 MB, so we need to up that. The maximum on EOS 4.21 is 2 GB, so that's what I'm going to use (and yes, the maximum value is 2,047 and not 2,048):

```
Arista(config-vm-GAD)#memory-size 2047
```

Remember, 2 GB of RAM from my 32 GB pool is not a big deal, but a large number of Arista switches out there have only 4 GB of RAM, so be very careful with your allocations here.

Next, I need to give the VM a network path so that I can connect to it. I'll be able to connect via the virtual console, as you'll see in a bit, but a switch is for networking, so let's give it a management interface. You can manually set a MAC address here if you'd like, but I'm just going to use the defaults and let EOS do the difficult work.

```
Arista(config-vm-GAD)#virtual-nic 1 Management1
```

Adding other interfaces requires VLAN Switch Virtual Interfaces (SVIs) to exist on the switch, which I learned the hard way (which is the same way I learn everything):

```
Arista(config)#vlan 101
Arista(config-vlan-101)#vlan 102
Arista(config-vlan-102)#virtual-machine GAD
Arista(config-vm-GAD)#virtual-nic 2 vlan 101
% Configuration ignored: Device Vlan101 does not exist.
```

It just won't work without an SVI. The SVI needs to exist, but it doesn't necessarily need to be configured:

```
Arista(config-vm-GAD)#int vlan 101
Arista(config-if-Vl101)# int vlan 102
Arista(config-if-Vl102)#virtual-machine GAD
Arista(config-vm-GAD)#virtual-nic 2 vlan 101
Arista(config-vm-GAD)#virtual-nic 3 vlan 102
```

Why do they need to be SVIs? My guess is that like most hypervisor environments, the assumption is that the VM will be a server and not a virtual switch, though it's completely fair to have a server that has an L2 trunk interface. At this point, I just accept this as a limitation of the wacky thing I'm trying to do and move on.

Before we go any further, I'm going to make a copy of the *vmdk* file (you'll see why in a moment):

```
Arista(config-vm-GAD-1)#copy flash:veOS-lab-4.21.1.1F.vmdk
vEOS2.vmdk
Copy completed successfully.
```

Finally, we should enable the VM:

```
Arista(config-vm-GAD)#enable
```

With those steps done, the VM should boot. You can check whether it's running by using the `show virtual-machine` command:

```
Arista#sho virtual-machine
VM Name           Enabled   State
-----
GAD-1             Yes       Running
```

To connect to the VM, use the `virtual-machine VM-Name console` command:

```
Arista#virtual-machine GAD-1 console
Connected to domain GAD-1
Escape character is
[ OK ]
Starting NorCal initialization: [ OK ]
Starting EOS initialization stage 2: [ OK ]
Starting Power On Self Test (POST): [ OK ]
Completing EOS initialization (press ESC to skip): [ OK ]
Model and Serial Number: unknown
System RAM: 2015524 kB
Flash Memory size: 4.0G

localhost login:
```

The VMs take a long time to boot due to resource constraints, but be patient and it will come up. You might need to disable ZTP, which you can do by using the `zerotouch disable` command (see [Chapter 13](#)). When the switch is done booting, you'll want to go in and change the hostname:

```
localhost login: admin
localhost>
localhost>en
localhost#conf
localhost(config)#hostname GAD-1
GAD-1(config)#
```

With the hostname configured, let's put an IP address on the management interface. Remember, this is a VM within an Arista switch, and the management interface of the VM is bound to the

management interface on the host switch. The management IP of the host switch is 10.0.0.1/24, so I'm going to use an IP from the same range that's not in use:

```
GAD-1(config)#int ma1
GAD-1(config-if-Ma1)#ip address 10.0.0.21/24
GAD-1(config-if-Ma1)#ping 10.0.0.100
PING 10.0.0.100 (10.0.0.100) 72(100) bytes of data.
80 bytes from 10.0.0.100: icmp_seq=1 ttl=64 time=0.000 ms
80 bytes from 10.0.0.100: icmp_seq=2 ttl=64 time=0.000 ms
80 bytes from 10.0.0.100: icmp_seq=3 ttl=64 time=0.000 ms
80 bytes from 10.0.0.100: icmp_seq=4 ttl=64 time=0.000 ms
80 bytes from 10.0.0.100: icmp_seq=5 ttl=64 time=0.000 ms

--- 10.0.0.100 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 8ms
rtt min/avg/max/mdev = 0.000/0.000/0.000/0.000 ms, ipg/ewma 2.000/0.000
ms
GAD-1(config-if-Ma1)#

```

NOTE

This is not actually a Layer 3 binding, so the IP address does not need to match the network on the host's management interface. I just like it that way to keep things simple.

Woohoo! I now have a VM running on a switch that's communicating through that switch to get to the network on the management interface. That's the easy part, though.

The Ethernet interfaces on our VM are bound to the SVIs for VLANs 101 and 102, so let's see how that works. The host switch has an IP address of 30.0.0.101/24 on VLAN 101, so let's see if we can do what we just did on the management interface and get Ethernet1 to work with an IP address on the same network. Actually, I'm going to make it

more complicated and add a VLAN 30 in the VM, and add an IP of 30.0.0.21/24 to the SVI. Here's the configuration on the VM named GAD-1:

```
GAD-1(config)#vlan 30
GAD-1(config-vlan-30)#int vlan 30
GAD-1(config-if-Vl30)#ip address 30.0.0.21/24
GAD-1(config-if-Vl30)#int e1
GAD-1(config-if-Et1)#switchport access vlan 30
GAD-1(config-if-Et1)#
```

Let's see if it works:

```
GAD-1(config-if-Et1)#ping 30.0.0.101
PING 30.0.0.101 (30.0.0.101) 72(100) bytes of data.

--- 30.0.0.101 ping statistics ---
5 packets transmitted, 0 received, 100% packet loss, time 52ms

GAD-1(config-if-Et1)#ping 30.0.0.102
PING 30.0.0.102 (30.0.0.102) 72(100) bytes of data.

--- 30.0.0.102 ping statistics ---
5 packets transmitted, 0 received, 100% packet loss, time 52ms
```

Well, crap. Let me save you the literal days of frustration I went through that I resolved only after talking to some kick-ass Arista developers who helped me out.

With vEOS running as a VM on an EOS switch, the interfaces within the VM don't behave the way you might think. To get this to work, you need to attach the IP address to the virtual bridge that is created in Bash within the vEOS instance. The `ifconfig` command from Bash is your friend here:

```
GAD-1(config-if-Et1)#bash
```

```
Arista Networks EOS shell
```

```
[admin@GAD-1 ~]$ ifconfig -a | grep et1
et1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 9214
vmnicet1: flags=4419<UP,BROADCAST,RUNNING,PROMISC,MULTICAST> mtu 1500
```

Remember, this is in the VM. Now, using the **ifconfig** command again we need to apply the same IP address to the **vmnicet1** bridge:

```
[admin@GAD-1 ~]$ sudo ifconfig vmnicet1 30.0.0.21/24
```

Now, we exit back to vEOS and test again:

```
[admin@GAD-1 ~]$ exit
logout
GAD-1(config-if-Et1)#ping 30.0.0.101
PING 30.0.0.101 (30.0.0.101) 72(100) bytes of data.
--- 30.0.0.101 ping statistics ---
5 packets transmitted, 0 received, 100% packet loss, time 52ms
```

Hmm. That's disheartening...

```
GAD-1(config-if-Et1)#ping 30.0.0.102
PING 30.0.0.102 (30.0.0.102) 72(100) bytes of data.
80 bytes from 30.0.0.102: icmp_seq=1 ttl=64 time=4.00 ms
80 bytes from 30.0.0.102: icmp_seq=2 ttl=64 time=0.000 ms
80 bytes from 30.0.0.102: icmp_seq=3 ttl=64 time=0.000 ms
80 bytes from 30.0.0.102: icmp_seq=4 ttl=64 time=0.000 ms
80 bytes from 30.0.0.102: icmp_seq=5 ttl=64 time=0.000 ms

--- 30.0.0.102 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 24ms
rtt min/avg/max/mdev = 0.000/0.800/4.000/1.600 ms, ipg/ewma 6.000/2.344
ms
```

Wait...what happened? The IP address 30.0.0.101 is on the host switch, and the VM cannot reach it. The IP address 30.0.0.102 is somewhere else on the network, and the VM *can* reach it. That appears to be a limitation of running vEOS in a VM within EOS (dizzy yet?).

In other words, the VM will be able to connect to devices other than the host, but not the host itself. This is also true on the management interface, though I kind of glossed over that earlier. The host's management IP is 10.0.0.1, and there is another host elsewhere on the network with the IP address of 10.0.0.100:

```
GAD-1#ping 10.0.0.1
PING 10.0.0.1 (10.0.0.1) 72(100) bytes of data.

--- 10.0.0.1 ping statistics ---
5 packets transmitted, 0 received, 100% packet loss, time 52ms

GAD-1#ping 10.0.0.100
PING 10.0.0.100 (10.0.0.100) 72(100) bytes of data.
80 bytes from 10.0.0.100: icmp_seq=1 ttl=64 time=0.000 ms
80 bytes from 10.0.0.100: icmp_seq=2 ttl=64 time=0.000 ms
80 bytes from 10.0.0.100: icmp_seq=3 ttl=64 time=0.000 ms
80 bytes from 10.0.0.100: icmp_seq=4 ttl=64 time=0.000 ms
80 bytes from 10.0.0.100: icmp_seq=5 ttl=64 time=0.000 ms

--- 10.0.0.100 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.000/0.000/0.000/0.000 ms, ipg/ewma 0.000/0.000
ms
```

This has to do with the way vEOS works and the fact that the interfaces in the VM are actually attached to virtual interfaces that you can't see within the VM. In fact, if you show the running configuration for the VM, you'll see that EOS (on the host) supplied MAC addresses for the interface bindings:

```
Arista#sho run sec virt
virtual-machine GAD-1
  disk-image flash:/veOS-lab-4.21.1.1F.vmdk
  memory-size 2047
  virtual-nic 1 Management1 52:54:00:02:d9:56
  virtual-nic 2 Vlan101 52:54:00:92:2b:60
  virtual-nic 3 Vlan102 52:54:00:f9:da:40
  enable
```

But, if we look at the MAC addresses in the VM (we'll use Bash), we can see that those MAC addresses are not the Ethernet interfaces in the VM but rather the `vmnicet#` interfaces that we had to configure in Bash:

```
[admin@GAD-1 ~]$ ifconfig et1 | grep ether
    ether 52:54:00:ad:fc:da  txqueuelen 500  (Ethernet)
[admin@GAD-1 ~]$ ifconfig vmnicet1 | grep ether
    ether 52:54:00:92:2b:60  txqueuelen 1000  (Ethernet)
```

As you can see, the configuration from the host is binding to the MAC address of the `vmnicet1` interface.

So, we have a VM running vEOS on our Arista switch. Pretty cool, and pretty weird, but it's not weird enough. Let's add another one! This will go much more quickly because I don't need to explain every step. I simply add another VM and call it something even more clever: GAD-2. Everything else is the same, but I use the copy of the vEOS `vmdk` file that we made.

```
Arista(config)#virtual-machine GAD-2
Arista(config-vm-GAD-2)#disk-image flash:vEOS-2.vmdk
Arista(config-vm-GAD-2)#memory-size 2047
Arista(config-vm-GAD-2)#virtual-nic 1 Management1
Arista(config-vm-GAD-2)#virtual-nic 2 Vlan101
Arista(config-vm-GAD-2)#virtual-nic 3 Vlan102
Arista(config-vm-GAD-2)#enable
Arista(config-vm-GAD-2)#exit
Arista(config)#
```

We cannot use the same `vmdk` file that we did on GAD-1 because that will result in two VMs using the same image. Although some operating systems can work that way, you don't want to do that with vEOS.

We should now have two VMs running:

```
Arista(config)#sho virtual-machine
VM Name           Enabled   State
-----
GAD-1             Yes       Running
GAD-2             Yes       Running
```

WARNING

By the way, if this isn't obvious by now, my two VMs are now consuming 4 GB of RAM. That's how much memory many Arista switches have in total, so you shouldn't be doing this on your very important production switches in the middle of the day. If you are, though, you've probably stopped reading because the network is on fire.

With the new VM running (remember, it will take a while to boot), let's go in and set the hostname and management IP address:

```
Arista(config)#virtual-machine GAD-2 console
Connected to domain GAD-2
Escape character is

localhost login: admin
localhost>
localhost>en
localhost#conf
localhost(config)#hostname GAD-2
GAD-2(config)#
```

Oh, if you're following along on a real switch, you might have noticed that **control-c** will bounce you out of the VM, so if you're used to doing that to get out of configuration mode, you're going to get quite annoyed. At this point, I am quite annoyed. Bad habits are hard to break.

Time to set the management IP address:

```
GAD-2(config)#int ma1
GAD-2(config-if-Ma1)#ip address 10.0.0.22/24
GAD-2(config-if-Ma1)#ping 10.0.0.100
PING 10.0.0.100 (10.0.0.100) 72(100) bytes of data.
80 bytes from 10.0.0.100: icmp_seq=1 ttl=64 time=4.00 ms
80 bytes from 10.0.0.100: icmp_seq=2 ttl=64 time=0.000 ms
80 bytes from 10.0.0.100: icmp_seq=3 ttl=64 time=0.000 ms
80 bytes from 10.0.0.100: icmp_seq=4 ttl=64 time=0.000 ms
80 bytes from 10.0.0.100: icmp_seq=5 ttl=64 time=0.000 ms

--- 10.0.0.100 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 20ms
rtt min/avg/max/mdev = 0.000/0.800/4.000/1.600 ms, ipg/ewma 5.000/2.344
ms
```

Now let's configure VLAN 30 and Ethernet 1:

```
GAD-2(config-if-Ma1)#vlan 30
GAD-2(config-vlan-30)#int vlan 30
GAD-2(config-if-Vl30)#ip address 30.0.0.22/24
GAD-2(config-if-Vl30)#int e1
GAD-2(config-if-Et1)#switchport access vlan 30
```

Remember, that won't work until we also add the IP to the `vmnicet1` interface in Bash. Let's do that and then exit back to vEOS and test it:

```
[admin@GAD-2 ~]$ sudo ifconfig vmnicet1 30.0.0.22/24
[admin@GAD-2 ~]$ exit
logout
GAD-2(config-if-Et1)#
GAD-2(config-if-Et1)#ping 30.0.0.102
PING 30.0.0.102 (30.0.0.102) 72(100) bytes of data.
80 bytes from 30.0.0.102: icmp_seq=1 ttl=64 time=0.000 ms
80 bytes from 30.0.0.102: icmp_seq=2 ttl=64 time=0.000 ms
80 bytes from 30.0.0.102: icmp_seq=3 ttl=64 time=0.000 ms
80 bytes from 30.0.0.102: icmp_seq=4 ttl=64 time=0.000 ms
80 bytes from 30.0.0.102: icmp_seq=5 ttl=64 time=0.000 ms

--- 30.0.0.102 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 8ms
```

```
rtt min/avg/max/mdev = 0.000/0.000/0.000/0.000 ms, ipg/ewma 2.000/0.000
ms
```

Success! Remember that we can only ping off the box and not to the interfaces on EOS.

However...

This is my favorite part. With two VMs running on my Arista switch they can communicate *with each other* but not with the host switch!

The VM GAD-1 has an IP address of 30.0.0.21 on VLAN 30, whereas the VM GAD-2 has an IP address of 30.0.0.22 on VLAN 30. Check it out—GAD-2 can ping GAD-1:

```
GAD-2(config-if-Et1)#ping 30.0.0.21
PING 30.0.0.21 (30.0.0.21) 72(100) bytes of data.
80 bytes from 30.0.0.21: icmp_seq=1 ttl=64 time=4.00 ms
80 bytes from 30.0.0.21: icmp_seq=2 ttl=64 time=0.000 ms
80 bytes from 30.0.0.21: icmp_seq=3 ttl=64 time=0.000 ms
80 bytes from 30.0.0.21: icmp_seq=4 ttl=64 time=4.00 ms
80 bytes from 30.0.0.21: icmp_seq=5 ttl=64 time=0.000 ms

--- 30.0.0.21 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 28ms
rtt min/avg/max/mdev = 0.000/1.600/4.001/1.960 ms, ipg/ewma 7.000/2.783
ms
```

It cannot ping the host in which it resides:

```
GAD-2(config-if-Et1)#ping 30.0.0.101
PING 30.0.0.101 (30.0.0.101) 72(100) bytes of data.

--- 30.0.0.101 ping statistics ---
5 packets transmitted, 0 received, 100% packet loss, time 52ms
```

But it can ping hosts other than the host on which it resides!

```
GAD-2(config-if-Et1)#ping 30.0.0.102
```

```
PING 30.0.0.102 (30.0.0.102) 72(100) bytes of data.  
80 bytes from 30.0.0.102: icmp_seq=1 ttl=64 time=0.000 ms  
80 bytes from 30.0.0.102: icmp_seq=2 ttl=64 time=0.000 ms  
80 bytes from 30.0.0.102: icmp_seq=3 ttl=64 time=0.000 ms  
80 bytes from 30.0.0.102: icmp_seq=4 ttl=64 time=0.000 ms  
80 bytes from 30.0.0.102: icmp_seq=5 ttl=64 time=0.000 ms  
  
--- 30.0.0.102 ping statistics ---  
5 packets transmitted, 5 received, 0% packet loss, time 8ms  
rtt min/avg/max/mdev = 0.000/0.000/0.000/0.000 ms, ipg/ewma 2.000/0.000  
ms
```

I think that's crazy and cool and all sorts of weird. Does it have any practical applications? Perhaps. I'll leave that to your imagination, but the fact that it's possible is one of those things that, to me at least, just shows how freaking cool EOS is.

By the way I feel that I should point out that the VLAN is not being tagged and that the VLAN numbers in each of my VMs do not need to match. Interface Et1 in those VMs is not a trunk, and the VLAN is locally significant. If you don't believe me, try it for yourself with different VLANs in each VM.

Hey, I wonder if we could add a VM to the vEOS VM running in EOS?

```
GAD-2#virt?  
virtual-machine not supported on this hardware platform
```

Heh—I guess that's enough *Inception*-level configuration madness for today.

If you'd like to see detail about your VMs, you can use the `show virtual-machine detail` command:

```

Arista#sho virtual-machine detail
Virtual Machine: GAD-1
  Enabled:          Yes
  State:           Running
  Disk Image:      /mnt/flash/veOS-lab-4.21.1.1F.vmdk
  Memory Size:    2047MB
  VNC port:        None
  Virtual Nic: vnic1
    Mac Address: 52:54:00:02:d9:56
    Device:       Management1
    Model Type:  e1000
  Virtual Nic: vnic2
    Mac Address: 52:54:00:92:2b:60
    Device:       Vlan101
    Model Type:  e1000
  Virtual Nic: vnic3
    Mac Address: 52:54:00:f9:da:40
    Device:       Vlan102
    Model Type:  e1000

Virtual Machine: GAD-2
  Enabled:          Yes
  State:           Running
  Disk Image:      /mnt/flash/vEOS-2.vmdk
  Memory Size:    2047MB
  VNC port:        None
  Virtual Nic: vnic1
    Mac Address: 52:54:00:b8:b4:a6
    Device:       Management1
    Model Type:  e1000
  Virtual Nic: vnic2
    Mac Address: 52:54:00:7e:4c:e1
    Device:       Vlan101
    Model Type:  e1000
  Virtual Nic: vnic3
    Mac Address: 52:54:00:23:e2:ad
    Device:       Vlan102
    Model Type:  e1000

```

Let's shut down those VMs so that we can free up all that memory:

```

Arista#conf
Arista(config)#virtual-machine GAD-1
Arista(config-vm-GAD-1)#no enable
Arista(config-vm-GAD-1)#virtual-machine GAD-2
Arista(config-vm-GAD-2)#no enable

```

```
Arista(config-vm-GAD-2)#^z
Arista#sho virtual-machine
  VM Name          Enabled   State
  -----          -----   -----
  GAD-1           No       Stopped
  GAD-2           No       Stopped
```

If you leave them enabled and `write` the configuration, they will automatically start when the switch reboots. With them disabled, the only resources they consume are the fairly large amount of flash disk space being consumed:

```
Arista#dir vEOS*
Directory of flash:/vEOS*
-rwx 725680128          Feb 14 20:44  vEOS-2.vmdk
-rwx 725614592          Feb 14 21:03  vEOS-lab-4.21.1.1F.vmdk

3440762880 bytes total (578560000 bytes free)
```

If I were to do something like this in a real network, I'd probably make sure to use a switch with a solid-state drive, which will probably be available if you have switches with 32 GB of RAM. The flash drive is a precious commodity, so you should use it sparingly.

Conclusion

So, what can you do with vEOS? You can build a Bow Tie MLAG (man, how I hate that term) environment. You can do routing tests. You can play around with ZTP, or EOS Application Programmable Interface (eAPI), or anything else that doesn't require a hardware ASIC. What requires an ASIC? Things covered in this book include tap aggregation, latency analyzer (LANZ), and FlexRoute, but there are certainly other features that do, too.

My favorite thing to use vEOS for is testing eAPI scripts. With a single VM running and the ability to connect to it using IP from my Mac I can test all sorts of crazy stuff without ever leaving the virtual environment on my laptop. I've also written entire class modules sitting on a plane with vEOS running. The only downside of that is the CPU utilization, which consumes battery and makes my lap hot, but that's kind of a *me problem* that doesn't affect people who are smart enough to put their laptops on the little tray that's there pretty much for that very purpose.

Chapter 33. Aristacisms

This chapter is an appendix of sorts, but it's not really, because some of the content comprises items that I felt should be included in the book, but weren't long enough to warrant their own chapter, whereas other items are just Arista-specific terms or nuances that I felt should be included but didn't fit in any other chapter. This chapter is like the trash bin in my brain. I took a lot of notes for topics to put in this book, and this is the stuff that's left over or didn't make it into other chapters.

It is with great pleasure that a writer makes up his own words, and because I've been writing for a year and this is the last chapter, I'm a little punchy. I do hereby dub the items in this chapter, *Aristacisms*.

Arista-Specific Configuration Items

Although Arista's Extensible Operating System (EOS) is a decidedly open source platform, there are some things that are configured differently on an Arista switch than on other vendor's switches. This is my repository for such items that don't warrant their own chapters.

There Is No Duplex Statement in EOS

I learned this the hard way when I tried to import 20 or so Cisco switch configurations into my client's shiny new Arista switches. Many of the ports had hardcoded speed and duplex settings (which I quickly scolded them for; read [Chapter 3](#) in *Network Warrior* if you don't

know why). Every one of these entries failed when I pasted them into EOS. Here's what happened when I tried:

```
SW1(config-if-Et5)#Speed 1000
% Invalid input
SW1(config-if-Et5)#Duplex full
% Invalid input
```

Flummoxed, I scrambled around quickly within EOS until I discovered that the **speed** command is supported (**duplex** is not), but it works in a very different fashion than that which I was used to.

On a 7050TX switch (48-port copper with four Quad Small Form-Factor Pluggables [QSFPs]), the following options are available for the interface **speed** command:

```
7050TX(config-if-Et33)#speed ?
 100full      Disable autoneg and force 100 Mbps/full duplex operation
 10full       Disable autoneg and force 10 Mbps/full duplex operation
 auto         Enable autoneg for speed, duplex, and flowcontrol
 forced        Disable autoneg and force speed/duplex/flowcontrol
 sfp-1000baset Configure autoneg and speed/duplex on 1000BASE-T SFP

7050TX(config-if-Et33)#speed forced ?
 10000full    Disable autoneg and force 10 Gbps/full duplex operation
 1000full     Disable autoneg and force 1 Gbps/full duplex operation
 1000half     Disable autoneg and force 1 Gbps/half duplex operation
 100full      Disable autoneg and force 100 Mbps/full duplex operation
 100gfull     Disable autoneg and force 100 Gbps/full duplex operation
 100half      Disable autoneg and force 100 Mbps/half duplex operation
 10full       Disable autoneg and force 10 Mbps/full duplex operation
 10half       Disable autoneg and force 10 Mbps/half duplex operation
 25gfull     Disable autoneg and force 25 Gbps/full duplex operation
 40gfull     Disable autoneg and force 40 Gbps/full duplex operation
 50gfull     Disable autoneg and force 50 Gbps/full duplex operation
```

Resisting the urge to configure the port for 50 Gbps/full duplex, I used the following command to result in a hardcoded 1 Gbps/full duplex

interface:

```
7050TX(config-if-Et2)#speed forced 1000full
```

Even though much of the configuration can be cut and pasted from other vendors' switches, beware of speed and duplex when they're hardcoded. Better yet, convince the server guys to stop hardcoding gigabit interfaces.

Watch Out for Those Comments!

One of my favorite features of EOS is the ability to put comments in the configuration. As you've no doubt seen by now, when a comment is allowed, I'll jam one in there. Blame my obsessive nature and my college professor who insisted on well-commented code while I was programming COBOL on punch cards in 1984. Ooh, 80s flashback time!

Anyway, another case of pasting configurations from an IOS device bit me in the ass when, after a few days, I got a call about one of the ports on the new Arista switches not working. After digging in, one of the guys on the team discovered this configured on the port:

```
interface Ethernet25
!
interface Ethernet26
!
! interface Ethernet28
description Server11
switchport access vlan 11
spanning-tree portfast
!
interface Ethernet27
!
interface Ethernet28
```

```
!
```

Anything look wrong there to you?

This is the relevant snippet of what was actually pasted:

```
interface Ethernet26
    description Server10
    switchport access vlan 10
    spanning-tree portfast
!
interface Ethernet28
    description Server11
    switchport access vlan 11
    spanning-tree portfast
!
```

When I pasted the configuration, I did so with a console cable, and, apparently, I pasted too much, which caused the serial port buffer to overrun. That caused a glitch, which caused a carriage return to be missed, which caused the line containing the command `interface Ethernet28` to be considered part of the comment line before it. Because interfaces can include comments, it just took that line as a comment on the previously configured interface (I had removed all empty interfaces to make the cut/paste job go more quickly). As a result, this is what the command parser saw, which resulted in the configuration shown previously:

```
interface Ethernet26
    description Server10
    switchport access vlan 10
    spanning-tree portfast
!
!interface Ethernet28
    description Server11
    switchport access vlan 11
    spanning-tree portfast
```

When pasting configurations from one switch into another, verify what you've pasted. If possible, don't use a console cable. And whatever you do, watch out for those comments!

The cool thing that I learned from all this was that I could put comments into interface configurations, so now I can annoy my clients with interface descriptions *and* comments!

Some Routing Protocols Are Shut Down by Default

When you configure a simple Routing Information Protocol (RIP) configuration, before you spend hours trying to figure out why it doesn't work, issue the `show active` command. You'll promptly discover that RIP is in a shutdown state by default:

```
SW1(config-router-rip)#sho active
router rip
  ! - RIPv2 link to R1
  network 10.0.0.1/32
  shutdown
```

Clearly, even the switch thinks you should use something else, but if you're hell-bent on using RIP, negating the `shutdown` command will bring up RIP:

```
SW1(config-router-rip)#no shut
```

Honestly, by the time most of you will read this, we'll be into the 2020s, so I think it's time to stop using RIP.

Trunk Groups

Trunk groups are mentioned in the Multichassis Link Aggregation

(MLAG) chapter ([Chapter 18](#)), but they're worth another mention here because they're pretty darn cool. Here's what the EOS 4.21.3F configuration guide has to say about them:

A trunk group is the set of physical interfaces that comprise the trunk and the collection of VLANs whose traffic is carried on the trunk. The traffic of a VLAN that belongs to one or more trunk groups is carried only on ports that are members of trunk groups to which the VLAN belongs, i.e., VLANs configured in a trunk group are pruned of all ports that are not associated with the trunk group.

I just love paragraphs like that in official documentation. They drive people to buy O'Reilly books.

A trunk group is a named group of VLANs with an added twist. Let's look at the potential for such a tool. We've already seen it in use for the MLAG peer, but let's try some other ways to use it. Here, I configure the range of VLANs 100 through 105 and then include them all in the trunk group `Leelu`:

```
Arista(config)#vlan 100-105
Arista(config-vlan-100-105)#trunk group Leelu
```

Now I'll go to interface `e10`, configure it as a trunk, and apply the trunk group `Servers` to that interface:

```
Arista(config-vlan-100-105)#int e1
Arista(config-if-Et1)#switchport mode trunk
```

Remember that, by default, all VLANs are included on trunks, but if we look at the output of `show vlan`, `Ethernet1` is not active for these

VLANs:

```
Arista(config-if-Et1)#sho vlan
```

VLAN	Name	Status	Ports
1	default	active	Et1
100	VLAN0100	active	
101	VLAN0101	active	
102	VLAN0102	active	
103	VLAN0103	active	
104	VLAN0104	active	
105	VLAN0105	active	

If we want the VLANs in a trunk group to be included on a trunk, we must add that trunk group to the interfaces in question:

```
Arista(config-if-Et1)#switchport trunk group Leelu
```

Now, the output of `show vlan` shows that all of these VLANs are active on the trunk port (ethernet 1):

```
Arista(config-if-Et1)#sho vlan
```

VLAN	Name	Status	Ports
1	default	active	Et1
100	VLAN0100	active	Et1
101	VLAN0101	active	Et1
102	VLAN0102	active	Et1
103	VLAN0103	active	Et1
104	VLAN0104	active	Et1
105	VLAN0105	active	Et1

Even though the ability to group VLANs like that is pretty cool, there's more to it than that. When a VLAN is added to a trunk group, trunk ports *must* include that trunk group in order to pass that VLAN! Using the same interface, I add the `switchport trunk allowed vlan`

command and then remove the `switchport trunk allowed vlan` command:

```
Arista(config-if-Et1)#switchport trunk allowed vlan 100-105
Arista(config-if-Et1)#no switchport trunk group Leelu
```

From experience, VLANs 100 through 105 should now be included on the e1 trunk, but the output of the `show vlan` command tells a different story:

```
Arista(config-if-Et1)#sho vlan
VLAN  Name                      Status    Ports
-----  -----
1      default                   active
100    VLAN0100                 active
101    VLAN0101                 active
102    VLAN0102                 active
103    VLAN0103                 active
104    VLAN0104                 active
105    VLAN0105                 active
```

To make matters worse, if you're not careful, you'll convince yourself that it should be working by looking at the output of the `show interface trunk` command:

```
Arista(config-if-Et1)#sho int trunk
Port      Mode      Status      Native vlan
Et1       trunk    trunking    1
Po1       trunk    trunking    1
Po1000   trunk    trunking    1

Port      Vlans allowed
Et1       100-105
Po1       All
Po1000   All

Port      Vlans allowed and active in management domain
Et1       None
Po1       None
```

Po1000	None
Port	Vlans in spanning tree forwarding state
Et1	None
Po1	None
Po1000	None

Take a look at the lines in bold in the previous output. It clearly says that VLANs 100 through 105 *are* allowed! The key is that they're not *allowed and active in the management domain* (next paragraph in the output). This is due to the fact that these VLANs are configured in a trunk group.

WARNING

You might be tempted to use the `trunk group` command as sort of a macro for a list of VLANs, and you can, but if you do that, the VLANs will be included in a trunk only if you use the trunk group. You can no longer include even one of the referenced VLANs in a trunk without using the trunk group. Think of this feature more as a way of securing VLANs from use as opposed to a way to make configurations simpler.

The good news is that VLANs can belong to more than one trunk group! This means that if we wanted to include only VLAN 101 on interface e11, we could do so by adding a new trunk group to VLAN 101 and then applying that trunk group to the interface. Let's try that.

First, I remove the `trunk allowed` stuff that wasn't working anyway:

```
Arista(config-if-Et1)#no switchport trunk allowed vlan 100-105
```

Now, I go to VLAN 101 and add a new trunk group. Remember, this VLAN is already part of trunk group `Servers`:

```
Arista(config-if-Et1)#vlan 101
Arista(config-vlan-101)#trunk group Multipass
```

At this point, VLAN 101 belongs to two trunk groups, **Leelu** and **Multipass**:

```
Arista(config-vlan-101)#int e31
Arista(config-if-Et31)#switchport mode trunk
Arista(config-if-Et31)#switchport trunk group Multipass
```

Now the output of **show vlan** is more to my liking:

```
Arista(config-if-Et31)#sho vlan
VLAN  Name                                Status    Ports
-----  -----
1      default                            active    Et1, Et31
100    VLAN0100                           active
101    VLAN0101                           active    Et31
102    VLAN0102                           active
103    VLAN0103                           active
104    VLAN0104                           active
105    VLAN0105                           active
```

You can also show what VLANs belong to what trunk groups by using the **show vlan trunk group** command:

```
Arista#sho vlan trunk group
VLAN      Trunk Groups
-----  -----
1
100      Leelu
101      Leelu, Multipass
102      Leelu
103      Leelu
104      Leelu
105      Leelu
```

Virtual Routing and Forwarding

EOS supports Virtual Routing and Forwarding (VRF) instances. VRFs are essentially multiple copies of the routing table that allow the same IP space to exist more than once in a single switch. Although Arista does support routing within the VRFs, it does not support something called *route leaking* or routing between VRFs.

NOTE

As this book is nearing its finishing stages, route leaking has been announced and is being supported in upcoming releases. Additionally, the structure of VRFs has changed such that the number of them supported is limited by memory and not the Application-Specific Integrated Circuits (ASIC) type.

You need to define VRFs by using the `vrf definition vrf-name` command mode. Although I'm not a fan of naming VRFs after colors, everyone does it, so I'm going to conform. Mostly.

```
Arista(config)#vrf definition PlumCrazy
Arista(config-vrf-PlumCrazy)#
```

The only thing that needs to be done here is to define a *route distinguisher*. VRFs in EOS work through something called *network namespaces* in Linux. This is a nifty operating system feature that is beyond the scope of this book, so I'll just let you know that the route distinguisher comprises two numbers separated by a colon (if you'd like to read up on how this really works, check out its page on the Arista website):

```
Arista(config-vrf-PlumCrazy)#rd ?
<admin>:<local assignment> Route Distinguisher

Arista(config-vrf-PlumCrazy)#rd 100:1
Arista(config-vrf-PlumCrazy)#
```

NOTE

This behavior has also changed in the latest revisions of EOS, and route discriminators may no longer be required when creating VRFs.

Because more VRFs equals more fun, let's add two more:

```
Arista(config-vrf-PlumCrazy)#vrf definition SubLime
Arista(config-vrf-SubLime)#rd 100:2
Arista(config-vrf-SubLime)#vrf definition ToxicOrange
Arista(config-vrf-ToxicOrange)#rd 100:3
```

Now that the VRFs are defined, we can apply them to interfaces. On this first interface I do what I do every single time and apply the IP address first:

```
Arista(config)#int vlan 101
Arista(config-if-Vl101)#ip address 11.0.0.2/24
Arista(config-if-Vl101)#vrf forwarding PlumCrazy
! Interface Vlan101 IP address 11.0.0.1 removed due to enabling VRF
PlumCrazy
Arista(config-if-Vl101)#
```

That's sort of a drag, but necessary I assume. No matter, I'll just reapply the IP address and all is well. Be prepared for this, because it will always remove the IP address if there's one assigned. It will also do this if you change or remove the VRF.

```
Arista(config-if-Vl101)#ip address 11.0.0.2/24
```

With that done, I need to add a default route for the new VRF. Routing is not permitted in the nondefault VRF, so all I can add is a static route, which is fine for management:

```
Arista(config)#ip route vrf PlumCrazy 0/0 11.0.0.1
```

Check out how I can add the same IP address to other interfaces in other VRFs. I can also add the same default route in each VRF:

```
Arista(config)#int vlan 102
Arista(config-if-Vl102)#vrf forwarding SubLime
Arista(config-if-Vl102)#ip address 11.0.0.2/24
Arista(config-if-Vl102)#ip route vrf SubLime 0/0 11.0.0.1
Arista(config)#int vlan 103
Arista(config-if-Vl103)#vrf forwarding ToxicOrange
Arista(config-if-Vl103)#ip address 11.0.0.2/24
Arista(config-if-Vl103)#ip route vrf ToxicOrange 0/0 11.0.0.1
Arista(config)#
```

To show how it all works, here's the default routing table with the default route in bold. Note the first line of the output that states VRF name: **default**:

```
Arista#sho ip route

VRF name: default
Codes: C - connected, S - static, K - kernel,
       0 - OSPF, IA - OSPF inter area, E1 - OSPF external type 1,
       E2 - OSPF external type 2, N1 - OSPF NSSA external type 1,
       N2 - OSPF NSSA external type2, B I - iBGP, B E - eBGP,
       R - RIP, I L1 - ISIS level 1, I L2 - ISIS level 2,
       O3 - OSPFv3, A B - BGP Aggregate, A O - OSPF Summary,
       NG - Nexthop Group Static Route, V - VXLAN Control Service

Gateway of last resort is not set

C      10.0.0.0/24 is directly connected, Management1
```

Here's the routing table for the other VRFs I created, with the different

default route:

```
Arista#sho ip route vrf PlumCrazy

VRF name: PlumCrazy
Codes: C - connected, S - static, K - kernel,
       O - OSPF, IA - OSPF inter area, E1 - OSPF external type 1,
       E2 - OSPF external type 2, N1 - OSPF NSSA external type 1,
       N2 - OSPF NSSA external type2, B I - iBGP, B E - eBGP,
       R - RIP, I L1 - ISIS level 1, I L2 - ISIS level 2,
       O3 - OSPFv3, A B - BGP Aggregate, A O - OSPF Summary,
       NG - Nexthop Group Static Route, V - VXLAN Control Service

Gateway of last resort:
  S      0.0.0.0/0 [1/0] via 11.0.0.1, Vlan101

  C      11.0.0.0/24 is directly connected, Vlan101

  ! IP routing not enabled
```

Crap! That's another mistake I make every single time: IP routing must be enabled on a per-VRF basis.

```
Arista(config)#ip routing vrf PlumCrazy
Arista(config)#ip routing vrf SubLime
Arista(config)#ip routing vrf ToxicOrange
```

Now let's see how the routing tables look. You can view them all by using the `sho ip route vrf all` command:

```
Arista(config)#sho ip route vrf all

VRF name: default
Codes: C - connected, S - static, K - kernel,
       O - OSPF, IA - OSPF inter area, E1 - OSPF external type 1,
       E2 - OSPF external type 2, N1 - OSPF NSSA external type 1,
       N2 - OSPF NSSA external type2, B I - iBGP, B E - eBGP,
       R - RIP, I L1 - ISIS level 1, I L2 - ISIS level 2,
       O3 - OSPFv3, A B - BGP Aggregate, A O - OSPF Summary,
       NG - Nexthop Group Static Route, V - VXLAN Control Service

Gateway of last resort is not set
```

```
C      10.0.0.0/24 is directly connected, Management1
```

VRF name: **PlumCrazy**

Codes: C - connected, S - static, K - kernel,
 0 - OSPF, IA - OSPF inter area, E1 - OSPF external type 1,
 E2 - OSPF external type 2, N1 - OSPF NSSA external type 1,
 N2 - OSPF NSSA external type2, B I - iBGP, B E - eBGP,
 R - RIP, I L1 - ISIS level 1, I L2 - ISIS level 2,
 03 - OSPFv3, A B - BGP Aggregate, A O - OSPF Summary,
 NG - Nexthop Group Static Route, V - VXLAN Control Service

Gateway of last resort:

```
S      0.0.0.0/0 [1/0] via 11.0.0.1, Vlan101
```

```
C      11.0.0.0/24 is directly connected, Vlan101
```

VRF name: **SubLime**

Codes: C - connected, S - static, K - kernel,
 0 - OSPF, IA - OSPF inter area, E1 - OSPF external type 1,
 E2 - OSPF external type 2, N1 - OSPF NSSA external type 1,
 [-- output truncated --]

With VRFs in place, I can ping using the `vrf vrf-name` keywords inserted directly after the `ping` command. I prefer the Cisco method that puts the VRF keyword at the end, because I constantly make this mistake, and just adding another keyword at the end is easier than inserting it into the middle of a command. But, after a few times of making the mistake, I've learned to adapt:

```
Arista#ping vrf PlumCrazy 11.0.0.1
PING 11.0.0.1 (11.0.0.1) 72(100) bytes of data.
80 bytes from 11.0.0.1: icmp_req=1 ttl=255 time=4.79 ms
80 bytes from 11.0.0.1: icmp_req=2 ttl=255 time=1.06 ms
80 bytes from 11.0.0.1: icmp_req=3 ttl=255 time=1.29 ms
80 bytes from 11.0.0.1: icmp_req=4 ttl=255 time=1.34 ms
80 bytes from 11.0.0.1: icmp_req=5 ttl=255 time=1.50 ms

--- 11.0.0.1 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 17ms
```

```
rtt min/avg/max/mdev = 1.069/2.001/4.791/1.402 ms, ipg/ewma
4.254/3.357 ms
```

When adding a VRF, the behavior of some common services changes a bit. For the most part, this involves the limitation that they can reside in only one VRF. Not surprisingly, these services are all related to management of the switch. Examples include Simple Network Management Protocol (SNMP), Syslog, TACACS+, Network Time Protocol (NTP), and DNS, all of which you should configure to work within the management VRF.

Configuring these services is simply a matter of adding the `vrf vrf-name` to the command, again, right after the command itself (not at the end). If you've previously configured these services, you will be greeted with a message similar to the following when configuring them for the new VRF:

```
Arista(config)#ip name-server 10.0.0.100
Arista(config)#ip name-server vrf PlumCrazy 11.0.0.100
% Nameservers are supported in only one VRF at a time.
Please unconfigure nameservers from Main VRF before continuing.
```

After you remove the other name servers, the VRF-enabled version no longer gives a warning:

```
Arista(config)#no ip name-server 10.0.0.100
Arista(config)#ip name-server vrf PlumCrazy 11.0.0.100
Arista(config)#
```

This is a great solution for switches that might reside in a customer's network but need to be managed from within the data center's management scheme.

Open Flow and Direct Flow

Open Flow is the ability for certain Arista switches to have their forwarding tables programmed remotely. Direct Flow is the same thing, but configurable through the command-line interface (CLI) on the switch itself. A clever application of this technology is the [Direct Flow Assist Solution brief](#). This basically allows the Palo Alto firewall to tell the Arista switch that a flow is allowed and thus offloads the traffic to the switch so that it can traverse the network more quickly and without the added latency that the firewall incurs. Open Flow and Direct Flow are hardware features and are thus only supported on certain hardware.

Macro-Segmentation Service

Macro-Segmentation Service (MSS) is the ability to program a supported firewall into the network wherever you want. Using a combination of CloudVision, Direct Flow, Virtual eXtensible Local Area Network (VXLAN), and some very clever thinking, MSS allows the ability to basically place a firewall logically in a way that would have caused physical redesigns, outages, and additional expense in a traditional network.

And, Finally...

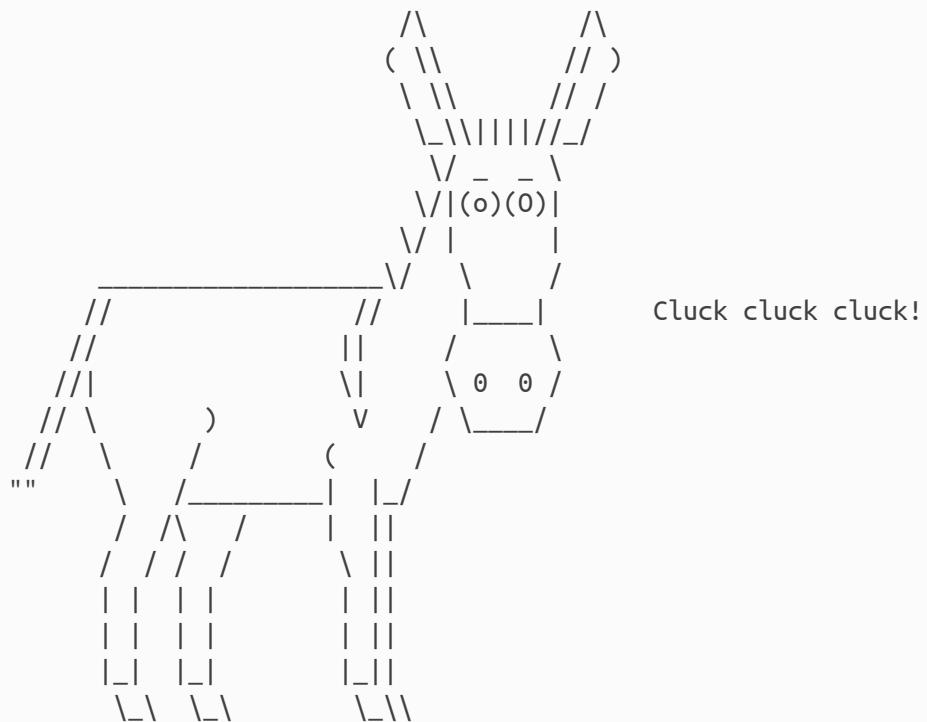
Because you wouldn't believe me unless I showed you myself, here is the version I'm running first:

```
Arista#sho ver | inc Software
Software image version: 4.21.1F
```

For your Arista CLI enjoyment:

Arista#**sho donkeys**

```
Farm utilization for 5 seconds: 0%/0%; 1 minute: 0%; 5 minutes: 0%
DID S Ty      DC Runtime(ms) Rides Poops      Hay DKY Donkeyname
 1 M sp 602F3AF0          0  1627      0 2600/3000  0 Eeyore
 2 F we 60C5BE00          4   136      29 5572/6000  0 Tingaleo
 3 F st 602D90F8         1676   837  2002 5740/6000  0 Daisy
 4 M we 602D08F8          0     1      0 5568/6000  0 Wonky Don
 5 F we 602DF0E8          0     1      0 5592/6000  0 Dakota
 6 M st 60251E38          0     2      0 5560/6000  0 Superdonkey
 7 M we 600D4940          0     2      0 5568/6000  0 Cookie Dough
 8 F we 6034B718          0     1      0 2584/3000  0 Sandy
 9 F we 603FA3C8          0     1      0 5612/6000  0 Kekie
10 M we 603FA1A0          0   8124      0 5488/6000  0 Shrek
11 M we 603FA220          0     9      0 4884/6000  0 BillyJoe-Bob
12 U we 60406818         124  2003     61 5300/6000  0 Smokey
13 M we 60581638          0     1      0 5760/6000  0 Snickers
14 M we 605E3D00          0     2      0 5564/6000  0 D.K.
15 M we 605FC6B8          0     2  011568/12000  0 Hee-Haw
```



It works on every version I've tried it on, as does **show chickens**.
show farm shows them both (they're technically different)! But wait,

there's more! Check out what happens if you run `show elephants`:

Arista#show elephants

```
% Internal error
% To see the details of this error, run the command 'show error 0'
Arista#
```

Uh-oh! I broke it! Better check out that error:

```
Arista#sho error 0  
Command: show elephants
```

```
===== Exception raised in 'Cli [interac -d -i --dlopen
-p -f -l libLoadDynamicLibs.so procmgr libProcMgrSetup.so
--daemonize' (PID 4951; PPID 4950) =====
Local variables by frame (innermost frame last):
```

[-- much scary output removed because it's scary --]

```
  File "/usr/lib/python2.7/site-packages/BasicCli.py", line 2107,
in runShowCommand
    result = valueFunction( mode, **kwargs )
  File "/usr/lib/python2.7/site-packages/CliPlugin/DonkeyCli.py",
line 103, in showElephants
    ''')
ScaredyException:
```

ScaredyException:

(nnn_. ' -nnn- ' _(|`
Try that on another vendor's switch! Why do these commands exist?
That's not for me to say, but I love them.

Conclusion

Hopefully, after all these pages, you have the same sense of how great Arista is as I do. It's a great place to work, it has great people at all levels who make and support great products, and it is fun in a way that permits things like `show elephants` to remain in the code. After almost 10 years of exposure and more than six years of employment, I can honestly say that Arista is still the best and most innovative networking company I've ever been exposed to.

I wrote the first edition of *Arista Warrior* because the company and the technology blew me away. I wrote the second edition for the same reasons. I'm excited to see what Arista produces over the next few years to warrant a third.

Index

SYMBOLS

#! (shebang), [Booting with ZTP](#)

\$ (dollar sign), with interface ranges, [Using EOS](#)

\$INTF, [on-intf](#)

\$IP-PRIMARY, [on-intf](#)

\$IP-SECONDARY, [on-intf](#)

\$IP6-PRIMARY, [on-intf](#)

\$IP6-SECONDARY, [on-intf](#)

\$OPERSTATE, [on-intf](#)

.schema command, SQLite, [ARP](#), [MAC](#)

/mnt/flash directory, [Some Quick EOS Bash Tips](#), [Aboot](#), [Aboot](#), [Configuring Event Monitor](#)

? (question mark), for command list, [Using EOS](#)

| (vertical bar), for piping, [Using EOS](#)

A

AAA (Authentication, Authorization, and Accounting), [Quick Things to Know](#), [Configuring Event Handlers](#)

Aboot, [Aboot-Conclusion](#)

boot-config, [Upgrading EOS](#), [Aboot](#)

conditions for halting boot, [Aboot](#)
entering shell, [Aboot](#)
list of commands, [Aboot](#)
password recovery, [Password Recovery](#)

ACL (Access-Control List), [Filtering Advanced Mirroring Sessions](#)-
[Filtering Advanced Mirroring Sessions](#)

actions

Bash, [on-startup-config](#)
event handler, [Description](#)

Address Resolution Protocol (see ARP)

advanced mirroring, [Advanced Mirroring](#)-[Conclusion](#)

about, [Advanced Mirroring](#)-[Advanced Mirroring](#)
filtering sessions, [Filtering Advanced Mirroring Sessions](#)-[Filtering Advanced Mirroring Sessions](#)

truncation with, [Truncation with Advanced Mirroring](#)

advertisement interval, [VRRP](#), [Basic Configuration](#)

agent command, [SysDB](#)

airflow, [Airflow](#)

AlgoMatch, [AlgoMatch](#)

AnyCloud, [AnyCloud](#), [AnyCloud](#)

Application-Specific Integrated Circuits (see ASICs)

arbitration, [Buffers](#)-[Buffers](#)

Arista

growth of, [My Personal Take](#)

history of, [A Brief History of Arista](#)-[Jayshree Ullal](#)

product portfolio, [Arista Products](#)-[Conclusion](#)

software products, [Software Products](#)-[CVX](#)

support from, [Arista Support](#)

switch features, [Arista Delivers](#)

(see also switches)

switches, [Switches](#)-[Optics](#)

Arista 7010T, [Arista Product ASICs](#), [Power](#), [Nonblocking Architecture](#), [Using EOS](#), [Link Layer Discovery Protocol](#), [Containers in EOS](#)

Arista 7020T, [Power](#)

Arista 7048T, [Buffers](#)

Arista 7050QX, [Switch Names](#)

Arista 7050S-64, [Fabric Speed](#), [Power](#)

Arista 7050TX, [There Is No Duplex Statement in EOS](#)

Arista 7050TX-96, [Containers in EOS](#)

Arista 7150S, [Arista Product ASICs](#), [Arista Product ASICs](#), [Power](#), [USB](#), [Using EOS](#), [Queue Thresholds](#)

Arista 7280, [Queue Thresholds](#)

Arista 7280R, [SysDB](#), [Zero-Touch Provisioning](#), [Event Manager](#), [on-counters](#), [Performance Monitoring](#)

FlexRoute, [How FlexRoute Works](#)-[Conclusion](#)

LANZ and, LANZ, Queue Thresholds, Queue Thresholds, Queue Thresholds

merchant silicon and, The Debate

power supply, Power

rails, Rails

reinitializing, SysDB

routing tables and, Switches

tap aggregation and, Tap Aggregation-The TapAgg GUI

USB ports, Upgrading EOS

Arista 7280R-48C6, Upgrading EOS

Arista 7280R2, Arista Product ASICs

Arista 7280SE-72, Event Manager, on-counters, Performance Monitoring

Arista 7280SR-48C6, Arista and Merchant Silicon

Arista 7500, Fabric Speed

Arista 7500E, Fabric Speed, Fabric Speed

Arista 7500R

fabric speed, Fabric Speed, Fabric Speed

Leaf-Spine network, Leaf-Spine

routing tables and, Switches

tap aggregation, Tap Aggregation

Arista 7500R2, Switches

Arista 7508R, Using EOS

Arista DCS-7050TX-96-F, [Switch Names](#)

Arista DCS-7280SR-48C6-M-F, [How FlexRoute Works](#)-Conclusion

Arista FM6000-based switches, [Arista Product ASICs](#)

Arista Product Quick Reference Guide, [Switches](#)

Arista Trident-based switches, [Arista Product ASICs](#)

ARP (Address Resolution Protocol)

event monitoring, [ARP-ARP](#)

responding to requests (see [VARP](#))

arrays, Python, [Python Data Type Primer](#)-[Python Data Type Primer](#)

ASICs (Application-Specific Integrated Circuits), [Arista Product ASICs](#)-[Arista Product ASICs](#), [Why EOS Containers?](#)

(see also merchant silicon)

cascading, [Fabric Speed](#)

EOS support, [SysDB](#)

identifying type used in switch, [Arista Product ASICs](#)

LANZ, [Microbursts Visualized](#)

modular drivers for, [SysDB](#)

on switch fabrics, [Fabric Speed](#)

authentication

for email, [Email](#)

for VRRP, [Basic Configuration](#)

Authentication, Authorization, and Accounting (AAA), [Quick Things to Know](#), [Configuring Event Handlers](#)

autocompletion of commands, [Using EOS](#), [Using EOS](#)

automation, [Automation](#), [Automation](#)

B

backplane, [Fabric Speed](#), [Fabric Speed](#)

backup routers, [VRRP](#), [VRRP](#), [Basic Configuration](#)-[Basic Configuration](#)

Bash, [Bash](#), [Email](#), [Email](#)

Bash actions, [on-startup-config](#)

bash expr command, [Microbursts Visualized](#)

bash shell, [Bash](#)-[Conclusion](#)

EOS and, [EOS](#)

executing CLI commands from, [Bash](#)

limiting access to, [Bash](#)

starting, [Bash](#)

Bash wc command, [on-counters](#)

Bechtolsheim, Andy, [Andy Bechtolsheim](#), [Switches](#), [Optics](#)

BGP (Border Gateway Protocol), [MST Terminology](#)

configuration, [Using EOS](#)

tracing, [Tracing Agents \(Debugging\)](#)-[Tracing Agents \(Debugging\)](#), [Tracing Agents \(Debugging\)](#)

VXLAN and, [Configuring VXLAN](#)

blocking, [Buffers](#)

Bogon packets, [Route](#)

boot command, [Aboot](#), [Aboot](#)

boot console speed command, [Aboot](#)

boot secret command, [Aboot](#), [Aboot](#)

boot system command, [Upgrading EOS](#)

boot-config file, [Upgrading EOS](#), [Aboot](#), [Aboot](#), [Aboot](#)

bootloader for EOS (see [Aboot](#))

Border Gateway Protocol (see [BGP](#))

boundary ports, [MST](#), [MST](#)

broadcast storms, [VXLAN](#)

buffer bloat, [Buffers](#)

buffers, [Buffers](#)-[Conclusion](#)

analyzing performance of (see [LANZ](#))

latency increased by, [Buffers](#)

queue thresholds, [Queue Thresholds](#)-[Queue Thresholds](#)

size, [Buffers](#)

BUM traffic, [Understanding VXLAN](#)

Busybox, [Aboot](#)

C

C-suffix switches, [Switch Names](#)

C13-C14 connectors, [Power](#)

cat command, [Bash](#)

cd command, [Aboot, Aboot](#)

CDP (Cisco Discovery Protocol), [Link Layer Discovery Protocol](#)

cEOS (containerized EOS), [cEOS](#), [cEOS—EOS in a Container](#)-Some Things to Watch Out For

change control, [Change control](#)

channel-group command, [Configuring MLAG](#)

chassis switches, [Fabric Speed](#), [Switches](#), [Power](#), [USB](#)

checkpoint restore command, [Configuration Checkpoints](#)

checkpoints, configuration, [Configuration Checkpoints](#)-Configuration Checkpoints

Cheriton, David, [David Cheriton](#)

Cisco 3750, [Using EOS](#), [Link Layer Discovery Protocol](#), [Multiple Spanning Tree Protocol](#), [Queue Thresholds](#)

Cisco 6500s, [Fabric Speed](#)

Cisco 6509, [Arista Product ASICs](#), [Fabric Speed](#), [Fabric Speed](#)

Cisco Discovery Protocol (CDP), [Link Layer Discovery Protocol](#)

Cisco Etherchannel, [MLAG Overview](#)

Cisco GLBP, [VARP](#)

Cisco HSRP, [First-Hop Redundancy](#)

Cisco ISL, [Configuring MLAG](#)

Cisco MST, [MST](#)

Cisco Nexus 7000, [The Debate](#)

Cisco Nexus 7000s, [Understanding VXLAN](#)

Cisco Nexus 9500 series, Arista and Merchant Silicon

Cisco Nexus chassis models, Fabric Speed

Cisco Nexus N7K-M132XP-12L, Fabric Speed

Cisco OTV, VXLAN

Cisco PVST, Multiple Spanning Tree Protocol, MST

Cisco RPVST, Multiple Spanning Tree Protocol, MST

Cisco SFP, Optics

Cisco trunk encapsulation, Configuring MLAG

Cisco vPC, MLAG Consistency

CIST (Common Internal Spanning Tree), MST, MST, MST
Terminology

CLI (command line interface)

eAPI and, eAPI

Emacs control characters, Using EOS

Event Monitor commands, ARP

executing commands from bash, Bash

exiting, Bash

tap aggregation from, Tap Aggregation from the Command-Line
Interface-Tap Aggregation from the Command-Line Interface

Cli command, Bash

CLI Standalone mode, CLI Standalone Mode-Arista Support

Clos network, Leaf-Spine

Clos, Charles, Leaf-Spine

Cloud Tracer, [Cloud Tracer](#)

CloudVision, [CloudVision](#), [CloudVision](#), [CloudVision](#)-Conclusion

configuration replace command, [Configure Replace](#)

CVP, [CVP](#)

CVX, [CVX](#)

portal (see CVP)

products, [The CloudVision Family](#)

ZTP server, [Booting with ZTP](#)

CloudVision eXchange (see CVX)

CloudVision Portal (see CVP)

Cognitive Campus, [Cognitive Campus](#)

Cognitive Management Plane, [Cognitive Campus](#)

command line interface (see CLI)

commands

autocompletion, [Using EOS](#)

emailing output from (see email)

listing, [Using EOS](#)

piping, [Using EOS](#)-[Using EOS](#)

scheduling (see Scheduler)

sending in IMs (see CloudVision)

comments, in configurations, [Watch Out for Those Comments!](#)-[Watch Out for Those Comments!](#)

commit command, [Configuration Sessions](#)

Common and Internal Spanning Tree (see CIST)

Common Spanning Tree (CST) (see CST)

Compliance option, CVP devices, [Devices](#)

computational delay, [Buffers](#)

config checkpoint restore command, [Configuration Checkpoints](#)

config replace command, [SysDB](#)

config terminal command, [Using EOS](#)

configlet builder, [Configlet builder](#)

configlets, [Network Provisioning](#), [Configlets-Configlet builder](#)

configuration checkpoints, [Configuration Checkpoints-Configuration Checkpoints](#)

configuration management, [Configuration Management-Conclusion](#)

checkpoints, [Configuration Checkpoints-Configuration Checkpoints](#)

configure replace command, [Configure Replace-Configuration Checkpoints](#)

CVP, [Configuration management](#)

sessions, [Configuration Sessions-Configuration Sessions](#)

configuration merging, [Configure Replace-Configure Replace](#)

configuration sessions, [Configuration Sessions-Configuration Sessions](#)

configure replace command, [Configure Replace-Configuration Checkpoints](#)

CONSOLESPEED option, boot-config file, [Aboot](#)

container-manager command, [Containers in EOS](#)

Containerized EOS (see [cEOS](#))

containers, [Containers-Conclusion](#)

[cEOS, cEOS—EOS in a Container-Some Things to Watch Out For EOS](#), [Containers in EOS-Containers in EOS](#)

Control-A keystroke, [Using EOS](#)

Control-B keystroke, [Using EOS](#)

Control-C keystroke, [Aboot, Aboot, Aboot](#)

Control-E keystroke, [Using EOS](#)

Control-F keystroke, [Using EOS](#)

control-plane, [Configuring VXLAN, VXLAN with Manual Control-Plane-VXLAN with Manual Control-Plane](#)

Control-Z keystroke, [Using EOS](#)

CoPP filter, [Conclusion](#)

copy command, [Upgrading EOS](#)

copy installed-extensions boot-extensions command, [The EOS Extension System](#)

cp command, [Aboot, Aboot](#)

CPU utilization, [Performance Monitoring-Performance Monitoring](#)

CST (Common Spanning Tree), [MST, MST, MST Terminology, MST Terminology](#)

curly braces ({}), [Using EOS-Using EOS](#)

custom silicon

benefits and drawbacks, [The Debate](#)

defined, [The Debate](#)

CVP (CloudVision Portal), [CVP](#), [CloudVision Portal-Topology](#)

about, [CloudVision Portal](#)

apps, [CVP apps](#)

basics, [Quick Things to Know](#)-[Quick Things to Know](#)

changes in, [Quick Things to Know](#)

Cloud Tracer and, [Cloud Tracer](#)

configuration management, [Configuration management](#)

devices for, [Devices](#)

events, [Events](#)-[Events](#)

graphs, [Queue Thresholds](#)

login page, [Login Page](#)

Network Provisioning (see Network Provisioning)

streaming telemetry, [Streaming Telemetry](#)

topology, [Topology](#), [Topology](#)-[Topology](#)

CVX (CloudVision eXchange), [CVX](#), [CloudVision eXchange](#),

Devices, [VXLAN with CVX](#)-[VXLAN with CVX](#)

D

data center requirements, [The Needs of a Data Center](#)

data-plane, [Configuring VXLAN](#)

debugging, [Performance Monitoring](#)-[Performance Monitoring](#)

device rollbacks, [Rollback](#)

DHCP (Dynamic Host Configuration Protocol), [Zero-Touch Provisioning](#), [Booting with ZTP](#)

dictionary, Python, [Python Data Type Primer](#)

dir command, [Using EOS](#), [Upgrading EOS](#)

dir extension: command, [The EOS Extension System](#)

Direct Flow, [Open Flow and Direct Flow](#)

Direct Flow Assist Solution brief, [Open Flow and Direct Flow](#)

Docker, [cEOS—EOS in a Container](#)

Docker flags, [cEOS—EOS in a Container](#)

docker images command, [cEOS—EOS in a Container](#)

docker run -it command, [Containers in EOS](#)

docker run hello-world command, [cEOS—EOS in a Container](#)

docker stop containernname command, [cEOS—EOS in a Container](#)

dollar sign (\$), with interface ranges, [Using EOS](#)

dropped packets, [The Case for Low Latency, Buffers](#)

dual-primary detection, MLAG, [Dual-Primary Detection](#)

Duda, Ken, [Ken Duda, Software Products](#)

duplex statement, [There Is No Duplex Statement in EOS](#)

Dynamic Host Configuration Protocol (see DHCP)

E

eAPI (EOS Application Programmable Interface), [eAPI-Conclusion](#)

configuring, [Configuring eAPI](#)-[Configuring eAPI](#)

expect scripting, [Expect Scripting](#)-[Expect Scripting](#)

FlexRoute script with, [Simulating 800,000 Routes](#)-[Simulating 800,000 Routes](#)

Python data types, [Python Data Type Primer](#)-[Python Data Type Primer](#)

screen scraping, [Screen Scraping](#)-[Screen Scraping](#)

scripting with, [Scripting with eAPI](#)-[Scripting with eAPI](#)

web interface, [eAPI Web Interface](#)-[eAPI Web Interface](#)

ECMP (Equal-Cost MultiPathing), [Leaf-Spine](#)

Emacs control characters, [Using EOS](#)

email, [Email](#)-[Conclusion](#)

authentication, [Email](#)

configuring, [Email](#)-[Email](#)

debug information, [Email](#)

email server, [Email](#)

sending, [Email](#)-[Email](#)

TLS for, [Email](#)

email command, [Email](#), [Email](#)

enable command, [Using EOS](#)

encryption, [Using EOS](#)

end command, [Using EOS](#)

Enterprise, [What About the Enterprise?](#)

environment command, [SysDB](#)

EOS (Extensible Operating System), [EOS](#), [EOS](#), [Introduction to EOS](#)-[Conclusion](#)

bootloader (see [Aboot](#))

containers, [Containers in EOS](#)-[Containers in EOS](#)

displaying version of, [Upgrading EOS](#)-[Upgrading EOS](#)

EXEC mode, [Using EOS](#)

extensions, [The EOS Extension System](#)-[Conclusion](#)

logging in, [Using EOS](#)

other operating systems compared to, [Using EOS](#)-[Using EOS](#)

Privileged EXEC mode, [Using EOS](#)

SysDB and, [SysDB](#)-[SysDB](#)

tcpdump, [tcpdump in EOS](#)-[tcpdump in EOS](#)

upgrading, [Upgrading EOS](#)-[Conclusion](#)

vEOS in an EOS VM, [vEOS in an EOS VM](#)-[vEOS in an EOS VM](#)

EOS Application Programmable Interface (eAPI) (see [eAPI](#))

EOS Central website, [The EOS Extension System](#)

Equal-Cost MultiPathing (ECMP), [Leaf-Spine](#)

Esc-B keystroke, [Using EOS](#)

Esc-F keystroke, [Using EOS](#)

Etherchannel, [MLAG Overview](#)

Ethernet

front-panel interfaces, [Using EOS](#)

high-speed switches, [High-Speed Ethernet-High-Speed Ethernet](#)

Ethernet interface names, [Using EOS](#)

Ethernet VPN (EVPN), VXLAN with, [VXLAN with EVPN](#)

event handlers, [Description-on-startup-config](#)

configuring, [Configuring Event Handlers-Configuring Event Handlers](#)

status of, [Showing the Event Handler Status-Showing the Event Handler Status](#)

triggers for, [Description-Showing the Event Handler Status](#)

Event Manager, [Event Manager-Conclusion](#)

configuring event handlers, [Configuring Event Handlers-Configuring Event Handlers](#)

event handler status, [Showing the Event Handler Status-Showing the Event Handler Status](#)

triggers, [Description-on-startup-config](#)

Event Monitor, [Event Monitor-Conclusion](#)

ARP events, [ARP-ARP](#)

backing up logs for, [Configuring Event Monitor-Configuring Event Monitor](#)

configuration, [Configuring Event Monitor-Configuring Event Monitor](#)

disabling, [Configuring Event Monitor](#)

enabling, [Configuring Event Monitor](#)

IGMP snooping, [IGMP Snooping](#)

LACP, [LACP-LACP](#)

MAC events, [MAC-MAC](#)

maximum buffer size for, [Configuring Event Monitor](#)

MRoute, [MRoute](#)

routing table events, [Route-Route](#)

sqlite option, [Using Event Monitor](#)

versions, [A Note About Versions](#)

event-handler event-name command, [Configuring Event Handlers](#)

event-monitor all command, [Configuring Event Monitor](#)

event-monitor backup max-size command, [Configuring Event Monitor](#)

event-monitor backup path command, [Configuring Event Monitor](#)

event-monitor buffer max-size command, [Configuring Event Monitor](#)

event-monitor command, [Using Event Monitor](#)

events, CVP, [Events-Events](#)

EVPN (Ethernet VPN), VXLAN with, [VXLAN with EVPN](#)

exclamation point (!), [Watch Out for Those Comments!](#)

EXEC mode, [Using EOS](#)

exit command, [Using EOS, Bash](#)

expect scripting, [Expect Scripting-Expect Scripting](#)

Extensible Messaging and Presence Protocol (XMPP), [CloudVision](#)

Extensible Operating System (see EOS)

extension command, [The EOS Extension System](#)
extensions, EOS, [The EOS Extension System](#)-Conclusion
adding, [The EOS Extension System](#)-The EOS Extension System
list of, [The EOS Extension System](#)
making permanent, [The EOS Extension System](#)
removing, [The EOS Extension System](#)

F

fabric, [Buffers](#), [Fabric Speed](#)-Fabric Speed, [Understanding VXLAN](#)
fabric speed, [Fabric Speed](#)-Fabric Speed
fans, [Switch Names](#), [Upgrading EOS](#)
(see also airflow)
FastCLI command, [Bash](#)
FAT filesystems, [USB](#)
FAT32 filesystems, [USB](#)
Field Programmable Gate Array (FPGA), [Aboot](#)
FIFO (First In/First Out) buffers, [Buffers](#)
filtering advanced mirroring sessions, [Filtering Advanced Mirroring Sessions](#)-Filtering Advanced Mirroring Sessions
first hop redundancy, [First-Hop Redundancy](#)-Conclusion
VARP for, [VARP](#)-Configuring VARP
VRRP for, [VRRP](#)-Miscellaneous VRRP Stuff
First In/First Out (FIFO) buffers, [Buffers](#)

flags

Docker, cEOS—EOS in a Container

tcpdump, tcpdump in Linux

flash

checkpoints on, Configuration Checkpoints

configuration sessions on, Configuration Sessions

flash drive, Some Quick EOS Bash Tips

FlexRoute, FlexRoute, FlexRoute-Conclusion

configuring/using, Configuring and Using FlexRoute-Configuring and Using FlexRoute

route simulation lab, Simulating 800,000 Routes-Simulating 800,000 Routes

tracing agents, Tracing Agents (Debugging)

forced keyword, speed command, There Is No Duplex Statement in EOS

FPGA (Field Programmable Gate Array), Aboot

from-user command, email, Email

full duplex, Buffers

fullrecover command, Aboot

G

Gateway Load Balancing Protocol (GLBP), VARP

Gateway, VXLAN, VXLAN

GBIC standard, Optics

Generic Routing Encapsulation (GRE), [Advanced Mirroring](#)
global configuration mode, [Using EOS](#)
grep command, [Using EOS](#), [Using EOS](#), [Bash](#)
group number, VRRP, [VRRP](#)
GUI, TapAgg, [The TapAgg GUI](#)

H

HER (Head-End Replication), [Understanding VXLAN](#)
HOL (Head-of-Line) blocking, [Buffers](#)
HSRP (Hot Standby Router Protocol), [First-Hop Redundancy](#)
HTTP/HTTPS, eAPI and, [Configuring eAPI](#)-[Configuring eAPI](#)
Hybrid Cloud architecture, [AnyCloud](#)

I

ifconfig command, [Some Quick EOS Bash Tips](#), vEOS in an EOS VM
ifconfig command, Aboot, [Aboot](#), [Aboot](#)
IGMP (Internet Group Management Protocol) snooping, [IGMP Snooping](#)
image bundle, [Image management](#)
image management, Network Provisioning and, [Image management](#)
include command, [Using EOS](#)
instance command, [MST](#)
instances, MST, [MST](#), [MST Terminology](#)

Inter-Switch Link (ISL), [Configuring MLAG](#)

interact command, [A Note About Versions](#)

interface buffers (see buffers)

interface command, [Using EOS](#)

interface configuration mode, [Using EOS](#)

interface line protocol objects, [Basic Configuration](#)

interface names, [Using EOS](#), [tcpdump in Linux](#), [tcpdump in EOS](#)

Internal Spanning Tree (IST), [MST](#)

Internet Group Management Protocol (IGMP) snooping, [IGMP Snooping](#)

interswitch networks, vEOS and, [Building the Interswitch Networks](#), [Building the Interswitch Networks](#)

IP cloud, [Understanding VXLAN](#)

IP routing (see routing)

ip trigger, [on-intf](#)

ip virtual-router address command, [Configuring VARP](#)

ip6 trigger, [on-intf](#)

iperf command, [Queue Thresholds](#)

ISL (Inter-Switch Link), [Configuring MLAG](#)

IST (Internal Spanning Tree), [MST](#)

J

JSON, [Screen Scraping](#)

json library, [Scripting with eAPI](#)

K

key-value pairs, [Python Data Type Primer](#)

kill command, [SysDB](#)

killall command, [SysDB](#)

L

LACP (Link Aggregation Control Protocol)

active mode, [Configuring MLAG](#)

Event Monitor logging of events, [LACP-LACP](#)

LAG (Link Aggregation Group), [MLAG](#)

LANZ (latency analyzer), [Microbursts Visualized](#)-Conclusion

microbursts, [Microbursts Visualized](#)-[Microbursts Visualized](#)

queue thresholds, [Queue Thresholds](#)-[Queue Thresholds](#)

status of, [Queue Thresholds](#)

latency

buffers increasing, [Buffers](#)

low, [The Case for Low Latency](#)

latency analyzer (see LANZ)

Layer 2 (L2) network, [VXLAN](#)

Layer 3 (L3) protocol, [Leaf-Spine](#)

early nomenclature for, [FlexRoute](#)

MLAG and, Layer 3 with MLAG

Layer 3 Anycast Gateway (see VARP)

leaf switches, Leaf-Spine

Leaf-Spine network architecture, Leaf-Spine

leaking, route, Virtual Routing and Forwarding

Link Aggregation Control Protocol (see LACP)

Link Aggregation Group (LAG), MLAG

Link Layer Discovery Protocol (LLDP), Link Layer Discovery Protocol-Conclusion

Linux, EOS, SysDB

EOS and, Introduction to EOS

logs, Logs

tcpdump, tcpdump in Linux-tcpdump in Linux

list, Python, Python Data Type Primer, Scripting with eAPI

LLDP (Link Layer Discovery Protocol), Link Layer Discovery Protocol-Conclusion

lldp run command, Cisco, Link Layer Discovery Protocol

load - interval command, Queue Thresholds

login, EOS, Using EOS

logs, Logs

loops

blocked by STP, MST, MLAG Overview

pruning VLANs causing, Pruning VLANs with MST

low latency, [The Case for Low Latency](#)

ls command, [Aboot](#), [Aboot](#)

M

M-suffix switches, [Switch Names](#)

MAC events, monitoring, [MAC-MAC](#)

Macro-Segmentation Service (MSS), [Macro-Segmentation Service](#)

Martian packets, [Route](#)

master router, VRRP, [VRRP](#), [Basic Configuration](#)-[Basic Configuration](#)

merchant silicon, [Merchant Silicon](#)-[Conclusion](#)

Arista product ASICs, [Arista Product ASICs](#)-[Arista Product ASICs](#)

Arista's use of, [Arista and Merchant Silicon](#)-[Arista and Merchant Silicon](#)

benefits and drawbacks, [The Debate](#)

defined, [The Debate](#)

identifying type used in switch, [Arista Product ASICs](#)

merging, configuration, [Configure Replace](#)-[Configure Replace](#)

microbursts, [Buffers](#), [Microbursts Visualized](#)-[Microbursts Visualized](#)

mirror0, [Advanced Mirroring](#)

mirroring, advanced (see advanced mirroring)

MLAG (Multichassis Link Aggregation Group), [MLAG](#)-[Conclusion](#),
[VARP](#), [VXLAN with MLAG](#)-[VXLAN with MLAG](#)

about, [MLAG Overview](#)-[MLAG Overview](#)

Bowtie MLAG, [Bow Tie MLAG](#), [Bow Tie MLAG-Bow Tie MLAG config sanity](#), [MLAG Consistency-MLAG Consistency](#) configuring, [Configuring MLAG](#)-Configuring MLAG dual-primary detection, [Dual-Primary Detection](#) failover, [MLAG Failover-Dual-Primary Detection](#) managing, [Managing MLAG](#)-Managing MLAG spanning tree and, [Spanning Tree and MLAG](#)-Spanning Tree and MLAG upgrading, [MLAG In-Service Software Upgrade-Layer 3 with MLAG](#)

MLAG Domain, [MLAG Overview](#), [Configuring MLAG MLAG ISSU \(In-Service Software Upgrade\)](#), [Configuring MLAG MLAG In-Service Software Upgrade-Layer 3 with MLAG](#)

MLAG peer failure, [VXLAN with MLAG](#)

Mojo Networks, Inc., [Cognitive Campus](#)

monitoring performance, [Performance Monitoring-Turn It Off!](#)

monolithic code, [SysDB](#)

more command

Aboot, [Aboot](#)

EOS, [Using EOS](#)

MRoute events, [MRoute](#)

MSS (Macro-Segmentation Service), [Macro-Segmentation Service](#)

MST (Multiple Spanning Tree), [SysDB](#), [Multiple Spanning Tree](#)

Protocol-Conclusion

boundary ports, MST

instances of, MST-MST

pruning VLANs with, Pruning VLANs with MST-Pruning VLANs with MST

regions of, MST Terminology, MST Terminology-MST Terminology

terminology, MST Terminology-MST Terminology

MST0 (MST instance 0), MST, MST, MST-MST

Multichassis Link Aggregation Group (MLAG) (see MLAG)

Multiple Spanning Tree (see MST)

N

namespaces, network, Virtual Routing and Forwarding

naming conventions for Arista switches, Switch Names

NET commands, boot-config file, Aboot-Aboot

Network Attached Storage (NAS) protocol, Network-Based Storage

network designs, Network Designs-Conclusion

AnyCloud, AnyCloud

Bowtie MLAG, Bow Tie MLAG

Leaf-Spine architecture, Leaf-Spine

Spline architecture, Spline

Universal Spine, Universal Spine

VXLAN Overlay, Spline

Network Identifier, [VXLAN](#), [VXLAN](#)

network namespaces, [Virtual Routing and Forwarding](#)

Network Provisioning, [What CVP Is](#), [Network Provisioning-Rollback](#)

change control, [Change control](#)

configlets, [Configlets-Configlet builder](#)

image management, [Image management](#)

rollback, [Rollback-Rollback](#)

snapshots for, [Snapshots](#)

tasks, [Tasks-Tasks](#)

network rollbacks, [Rollback](#)

networking, data center, [Data Center Networking](#)

no autostate command, [Configuring MLAG](#)

no boot secret command, [Aboot](#)

no event-monitor all command, [Configuring Event Monitor](#)

no extension command, [The EOS Extension System](#)

no schedule command, [Scheduler](#)

no trace command, [Turn It Off!](#)

no vrrp group command, [Basic Configuration](#)

nonblocking switches, [Fabric Speed](#), [Fabric Speed-Fabric Speed](#), [Nonblocking Architecture](#)

north-south traffic, [Network Designs](#)

notification mode, [Queue Thresholds](#)

NT File System (NTFS), [USB](#)

nz command modifier, [Using EOS](#)

O

object identifiers (OID), [Streaming Telemetry](#)

on-boot, [Description](#)

on-boot command, [Containers in EOS](#)

on-counter trigger, [Configuring Event Handlers](#)

on-counters, [on-counters-on-counters](#)

on-intf, [Description](#), [on-intf](#)

on-logging, [Description](#), [on-logging](#)

on-maintenance, [Description](#), [on-maintenance](#)

on-startup-config, [Description](#), [on-startup-config](#)

on-startup-config trigger, [Configuring Event Handlers](#)

Open Flow, [Open Flow and Direct Flow](#)

operstatus trigger, [on-intf](#)

optics, [Optics-Optics](#)

OSPF (Open Shortest Path First), [Tracing Agents \(Debugging\)](#)

OTV (Overlay Transport Virtualization), [VXLAN](#), [Understanding VXLAN](#)

overlay, [VXLAN](#)

Overlay Network, [VXLAN](#), [VXLAN](#), [Understanding VXLAN](#)

P

packets, dropped, The Case for Low Latency, Buffers

Parkin, Rich, Using EOS, Queue Thresholds

PASSWORD command, Aboot, Aboot

password command, email, Email

PASSWORD option, boot-config file, Aboot

password recovery, Aboot, Password Recovery

passwords, EOS and, Using EOS

pcap file, tcpdump in Linux

peer-link, MLAG, MLAG Overview, Configuring MLAG-Configuring MLAG, MLAG Failover-MLAG Failover

Per-VLAN Spanning Tree (PVST), Multiple Spanning Tree Protocol, MST

performance monitoring, Performance Monitoring-Turn It Off!

pipes, Using EOS-Using EOS

PoE (Power Over Ethernet) switches, Switches

polling mode, Queue Thresholds, Queue Thresholds

power supplies, Power

preemption, VRRP, Basic Configuration, Basic Configuration

print command, Python, Python Data Type Primer, Scripting with eAPI

Privileged EXEC mode, Using EOS

process manager, SysDB-SysDB

processes

contained effects of crashing, SysDB-SysDB

killing, SysDB, SysDB-SysDB

propagation delay, Buffers

proprietary features, custom silicon and, Arista and Merchant Silicon

protocol http command, Configuring eAPI

protocol specific mode, Using EOS

ps -ef r command, Scheduler

pstree command, Some Quick EOS Bash Tips

PVST (Per-VLAN Spanning Tree), Multiple Spanning Tree Protocol, MST

pwd command, Bash

pwd command, Aboot, Aboot

Python

data types, Python Data Type Primer-Python Data Type Primer

FlexRoute script with, Simulating 800,000 Routes

print command, Python Data Type Primer, Scripting with eAPI

Q

Q-suffix switches, Switch Names

QoS (quality of service), LANZ

QSFP+ optics, Optics

question mark (?), for command list, Using EOS

queue thresholds, [Queue Thresholds](#)-[Queue Thresholds](#)

queue-monitor length command, [Queue Thresholds](#)

R

R-suffix switches, [Switches](#), [FlexRoute](#)

rails, [Rails](#)

Rapid Spanning Tree Protocol (RSTP), [MST](#)

Rapid-PVST (RPVST), [Multiple Spanning Tree Protocol](#), [MST](#)

regions, [MST](#), [MST Terminology](#)-[MST Terminology](#)

regular expressions, [Using EOS](#)

reload command, [Scripting with eAPI](#)

reload in time command, [Configuration Sessions](#)

reload now command, [MLAG In-Service Software Upgrade](#), [Scripting with eAPI](#)

reload-delay command, [MLAG Failover](#)

Rib agent, [Tracing Agents \(Debugging\)](#)-[Tracing Agents \(Debugging\)](#)

RIP (routing information protocol), [Tracing Agents \(Debugging\)](#)-[Tracing Agents \(Debugging\)](#), [Some Routing Protocols Are Shut Down by Default](#)

rollback clean-config command, [Configuration Sessions](#)

rollbacks

device, [Rollback](#)

network, [Rollback](#)

Network Provisioning, [Rollback](#)-[Rollback](#)

route distinguisher, management VRF, [Virtual Routing and Forwarding](#)
route leaking, [Virtual Routing and Forwarding](#)
routers

VRRP backup, [VRRP](#), [Basic Configuration](#)-[Basic Configuration](#)

VRRP master, [VRRP](#), [Basic Configuration](#)-[Basic Configuration](#)

routing information protocol (see RIP)

routing information protocol (RIP), [Tracing Agents \(Debugging\)](#)-
[Tracing Agents \(Debugging\)](#), [Some Routing Protocols Are Shut Down](#)
[by Default](#)

routing tables

event monitoring, [Route-Route](#)

R-suffix switches, [Switches](#)

RPMs, extending EOS using, [The EOS Extension System](#)-[Conclusion](#)

RPVST (Rapid-PVST), [Multiple Spanning Tree Protocol](#), [MST](#)

RSTP (Rapid Spanning Tree Protocol), [MST](#)

running-config, [Upgrading EOS](#)

running-config file, [Scripting with eAPI](#)-[Scripting with eAPI](#)

running-configs, [SysDB](#)

configuration replace command, [Configure Replace](#)

configuration sessions, [Configuration Sessions](#)

S

S-suffix switches, [Switch Names](#)

scalar variables, [Python Data Type Primer](#)

schedule command, [Scheduler](#), [Scheduler](#)

[Scheduler](#), [Scheduler](#)-Conclusion

creating scheduled job, [Scheduler](#)-Scheduler

deleting scheduled jobs, [Scheduler](#)

jobs names, case insensitivity of, [Scheduler](#)

log files for, [Scheduler](#)

viewing scheduled jobs, [Scheduler](#)

screen scraping, [Screen Scraping](#)-Screen Scraping

scripting

fear of, [GAD's Rant About the Fear of Scripting](#)

with eAPI, [Scripting with eAPI](#)-Scripting with eAPI

SDNs (Software Defined Networks), [Arista Product ASICs](#)

Segment, VXLAN, [Understanding VXLAN](#)

sequence, Python, [Python Data Type Primer](#)

server command, email, [Email](#)

service configuration checkpoint max saved number command, [Configuration Checkpoints](#)

set list command, [cEOS—EOS in a Container](#)

SFP+ optics, [Optics](#)

shape rate command, [Queue Thresholds](#)

shebang (#!), [Booting with ZTP](#)

sho event-handler trigger counters command, [on-counters](#)

sho ip route vrf all command, [Virtual Routing and Forwarding](#)

show active command, [Email](#), [Some Routing Protocols Are Shut Down by Default](#)

show agent names command, [Tracing Agents \(Debugging\)](#)

show boot command, [Upgrading EOS](#), [Aboot](#)

show cdp neighbors command, [Link Layer Discovery Protocol](#)

show clock command, [Bash](#)

show command, [email](#), [Email](#)

show config checkpoints command, [Configuration Checkpoints](#)

show container-manager info command, [Containers in EOS](#)

show cvx service Vxlan command, [VXLAN with CVX](#)

show donkeys command, [And](#), [Finally...](#)

show elephants command, [And](#), [Finally...](#)

show event-handler command, [Configuring Event Handlers](#)

show event-monitor arp command, [Using Event Monitor](#)

show event-monitor command, [Using Event Monitor](#)

show event-monitor lacp command, [LACP](#)

show event-monitor route command, [Route](#)

show extensions command, [The EOS Extension System](#)

show interface command, [LANZ](#)

show interface counters command, [Using EOS](#), [Microbursts Visualized](#)

show interface trunk command, [Trunk Groups](#)

show ip virtual-router command, [Configuring VARP](#)

show lldp command, [Link Layer Discovery Protocol](#)

show lldp neighbor, [cEOS—EOS in a Container](#)

show lldp neighbors command, [Link Layer Discovery Protocol](#)

show lldp neighbors detail command, [Link Layer Discovery Protocol](#)

show log command, [Using EOS](#)

show logging follow command, [Logs](#)

show mac address-table command, [VXLAN with Manual Control-Plane](#)

show management api http-commands command, [Configuring eAPI](#)

show management cvx status command, [VXLAN with CVX](#)

show mlag command, [Configuring MLAG](#), [Managing MLAG](#), [MLAG Failover](#), [MLAG Failover](#)

show mlag config-sanity command, [MLAG Consistency](#)-[MLAG Consistency](#), [Spanning Tree and MLAG](#)

show mlag detail command, [Managing MLAG](#), [MLAG Failover](#)-[MLAG Failover](#), [MLAG Failover](#), [Dual-Primary Detection](#)

show mlag interfaces command, [Managing MLAG](#)

show mlag issu compatibility command, [Configuring MLAG](#)-[Configuring MLAG](#)

show platform ? command, [Arista Product ASICs](#)

show platform jericho mapping command, [Queue Thresholds](#)

show process top command, [Performance Monitoring](#)-[Performance](#)

Monitoring

show queue-monitor length command, [Queue Thresholds](#)

show queue-monitor length csv command, [Queue Thresholds](#)

show queue-monitor length status command, [Queue Thresholds](#)

show run command, [Using EOS, Email](#)

show running-config command, [Using EOS, Using EOS](#)

show schedule command, [Scheduler](#)

show schedule summary command, [Scheduler](#)

show session-config command, [Configuration Sessions](#)

show spanning-tree command, [MST, MST, MLAG In-Service Software Upgrade](#)

show tap aggregation groups command, [Tap Aggregation](#)

show trace agent-name command, [Tracing Agents \(Debugging\)](#)

show track command, [Miscellaneous VRRP Stuff](#)

show version command, [Upgrading EOS-Upgrading EOS, Bash, MLAG Failover, Screen Scraping](#)

show vlan command, [Trunk Groups-Trunk Groups](#)

show vrrp brief command, [Miscellaneous VRRP Stuff](#)

show vrrp command, [Basic Configuration](#)

show vxlan address-table command, [VXLAN with Manual Control-Plane](#)

show vxlan controller status command, [VXLAN with CVX](#)

show vxlan flood vtep command, [VXLAN with Manual Control-Plane](#)

show vxlan vtep command, [VXLAN with Manual Control-Plane](#)

shutdown command, [SysDB](#)

silicon (see custom silicon; merchant silicon)

snapshots, [Network Provisioning](#), [Snapshots](#)

SNMP GET, [Microbursts Visualized](#)

Software Defined Networks (SDNs), [Arista Product ASICs](#)

software products, Arista, [Software Products-CVX](#)

AnyCloud, [AnyCloud](#)

cEOS, [cEOS](#)

CloudVision, [CloudVision](#)

EOS, [EOS](#)

switches, [Switches-Optics](#)

vEOS, [vEOS](#)

Spanning Tree Protocol (STP) (see STP)

spanning-tree mode command, [Multiple Spanning Tree Protocol](#)

spanning-tree mst priority command, [MST](#)

spanning-tree priority command, [MST](#)

spanning-tree root primary command, [MST](#)

speed command, [There Is No Duplex Statement in EOS](#)

spine switches, [Leaf-Spine](#)

Spine-Leaf network architecture, [Leaf-Spine](#)

Spline network architecture, [Spline](#)

split brain scenario, MLAG, Configuring MLAG, Dual-Primary Detection

(see also dual-primary detection)

SQLite, Using Event Monitor

sqlite command, A Note About Versions

SQLite option, Event Monitor, Using Event Monitor

stacked switches, The Needs of a Data Center

startup-config file, Password Recovery

replacing, SysDB

ZTP used in absence of, Zero-Touch Provisioning, Zero-Touch Provisioning

STP (Spanning Tree Protocol), SysDB, Multiple Spanning Tree Protocol

disabling, Configuring MLAG

loops blocked by, MLAG Overview

MLAG with, MLAG, Configuring MLAG

Stp agent, SysDB

streaming telemetry, Streaming Telemetry

sudo command, Logs, Containers in EOS

support from Arista, Arista Support

SVIs (Switch Virtual Interfaces), vEOS in an EOS VM

SWI files, Aboot

SWI option, boot-config file, Aboot

swiinfo command, [Aboot](#), [Aboot](#)

switch fabric

about, [Buffers](#)

speed of, [Fabric Speed](#)-[Fabric Speed](#)

switch speeds, [High-Speed Ethernet](#)-[High-Speed Ethernet](#)

Switch Virtual Interfaces (SVIs), [vEOS in an EOS VM](#)

switches

airflow for, [Airflow](#)

AlgoMatch, [AlgoMatch](#)

chassis, [Fabric Speed](#)

CVP, [Quick Things to Know](#)

Ethernet speeds for, [High-Speed Ethernet](#)-[High-Speed Ethernet](#)

fans for, [Airflow](#), [Upgrading EOS](#)

features, [Arista Delivers](#)

FlexRoute, [FlexRoute](#)

naming conventions for, [Switch Names](#)

nonblocking, [Fabric Speed](#), [Fabric Speed](#)-[Fabric Speed](#),
[Nonblocking Architecture](#)

optics for, [Optics](#)-[Optics](#)

power supplies for, [Power](#)

rails, [Rails](#)

requirements for, [The Needs of a Data Center](#)

stacked, The Needs of a Data Center

superfluous features on, The Needs of a Data Center

ToR, Fabric Speed, Why EOS Containers?

USB ports, USB-USB

switching, white-box, Why EOS Containers?

switchport tap command, Tap Aggregation

switchport trunk allowed vlan command, Trunk Groups

SysDB, SysDB, SysDB-Conclusion, SysDB-SysDB

effects of crashing, SysDB-SysDB

killing, SysDB

process state storage, SysDB-SysDB

T

TAC (Technical Assistance Center), Arista Support

tail command, Using EOS-Using EOS

tall -f filename command, Logs

tap aggregation, Tap Aggregation-Conclusion

configuring Arista 7280R for, Tap Aggregation-The TapAgg GUI

from CLI, Tap Aggregation from the Command-Line Interface-Tap Aggregation from the Command-Line Interface

TapAgg GUI, The TapAgg GUI

tap port, Tap Aggregation, Tap Aggregation

TapAgg GUI, The TapAgg GUI

TCAM (ternary content-addressable memory), [AlgoMatch](#)

tcpdump, [tcpdump](#) and [Advanced Mirroring-tcpdump in EOS](#)

EOS, [tcpdump in EOS](#)-[tcpdump in EOS](#)

Linux, [tcpdump in Linux](#)-[tcpdump in Linux](#)

tcpdump command, [Tap Aggregation from the Command-Line Interface](#)

Technical Assistance Center (TAC), [Arista Support](#)

telnet, [Using EOS](#)

ten gigabit switches, [High-Speed Ethernet](#)

TerminAttr, [Quick Things to Know](#), [Streaming Telemetry](#)

terminology, MST, [MST Terminology](#)-[MST Terminology](#)

timers, configuration session, [Configuration Sessions](#)

TLS (Transport Layer Security), [Email](#)

tool port, [Tap Aggregation](#)

top command, [Performance Monitoring](#)-[Performance Monitoring](#)

Top-of-Rack (ToR) switches, [Leaf-Spine](#), [Fabric Speed](#), [Why EOS Containers?](#)

trace agent-name command, [Tracing Agents \(Debugging\)](#)

trace command, [Tracing Agents \(Debugging\)](#)

trace monitor command, [Tracing Agents \(Debugging\)](#)

tracing

agents for, [Tracing Agents \(Debugging\)](#)-[Turn It Off!](#)

turning off, [Turn It Off!](#)

Transport Layer Security (TLS), [Email](#)

Trident 2 chipset, [Arista and Merchant Silicon](#)

triggers, event-handler, [Description-on-startup-config](#)

\$INTF, [on-intf](#)

\$IP-PRIMARY, [on-intf](#)

\$IP6-PRIMARY, [on-intf](#)

\$OPERSTATE, [on-intf](#)

Event Manager, [Description-on-startup-config](#)

for event handlers, [Description-on-startup-config](#)

ip trigger, [on-intf](#)

ip6 trigger, [on-intf](#)

on-counter trigger, [Configuring Event Handlers](#)

on-startup-config trigger, [Configuring Event Handlers](#)

operstatus trigger, [on-intf](#)

troubleshooting, [Troubleshooting-Conclusion](#)

CLI Standalone mode, [CLI Standalone Mode-Arista Support](#)

logs for, [Logs](#)

performance monitoring, [Performance Monitoring-Turn It Off!](#)

truncation, with advanced mirroring, [Truncation with Advanced Mirroring](#)

trunk encapsulation, [Configuring MLAG](#)

trunk group command, [Configuring MLAG, Trunk Groups](#)

trunk groups, [Trunk Groups](#)-[Trunk Groups](#)

Tunnel End Point, [VXLAN](#), [VXLAN](#), [VXLAN](#) with Manual Control-Plane

tunnels, [VXLAN](#), [Understanding VXLAN](#)

tuple, [Python](#), [Python Data Type Primer](#)

U

Ullal, Jayshree, [Jayshree Ullal](#)

uname command, [Bash](#)

underlay, [VXLAN](#)

Universal Spine network, [Universal Spine](#)

upstream link failure, [VXLAN with MLAG](#)

USB ports, [USB-USB](#)

user virtual address space, [SysDB](#)

username command, email, [Email](#)

V

VARP (Virtual ARP), [VARP](#)-Configuring VARP

configuration, [Configuring VARP](#)-Configuring VARP

shared virtual MAC address, [VARP](#), [Configuring VARP](#)

virtual IP address, [Configuring VARP](#)

VDC (Virtual Device Context), [Understanding VXLAN](#)

vendor lock, [Arista and Merchant Silicon](#)

vEOS, vEOS, vEOS-Conclusion

automating VirtualBox builds, Automating VirtualBox Builds

building interswitch networks, Building the Interswitch Networks-
Building the Interswitch Networks

creating base VM in VirtualBox, Creating the Base VM-Creating the Base VM

CVX and, CVX

making a real lab through cloning, Making a Real Lab Through Cloning-Making a Real Lab Through Cloning

running in an EOS VM, vEOS in an EOS VM-vEOS in an EOS VM

VirtualBox, VEOS in VirtualBox-Automating VirtualBox Builds

vEOS-Lab, vEOS, vEOS

vEOS-Router, vEOS, vEOS

vertical bar (|), for piping, Using EOS

veth pairs, Some Things to Watch Out For

VFAT filesystems, USB

vi command, Aboot, Aboot

Virtual ARP (see VARP)

Virtual Device Context (VDC), Understanding VXLAN

Virtual eXtensible Local Area Network (see VXLAN)

virtual IP (VIP), VRRP, Basic Configuration

virtual LAN (see VLAN)

virtual machines (see VM)

Virtual Output Queuing (VOQ), Buffers

Virtual Port Channel (vPC), MLAG, Bow Tie MLAG

virtual router identifier (VRID), VRRP

Virtual Router Redundancy Protocol (see VRRP)

Virtual Routing and Forwarding (VRF), Configuring eAPI, Virtual Routing and Forwarding-Virtual Routing and Forwarding

VirtualBox

automating builds, Automating VirtualBox Builds

building interswitch networks, Building the Interswitch Networks-
Building the Interswitch Networks

creating base VM, Creating the Base VM-Creating the Base VM

making a real lab through cloning, Making a Real Lab Through Cloning-Making a Real Lab Through Cloning

vEOS, VEOS in VirtualBox-Automating VirtualBox Builds

VLAN (virtual LAN)

for MLAG peer-to-peer communication, Configuring MLAG-
Configuring MLAG

MST for, MST, MST Terminology

pruning, Pruning VLANs with MST-Pruning VLANs with MST

trunk group, Trunk Groups-Trunk Groups

VM (virtual machines)

containers, Containers

CVP as, Quick Things to Know

VXLAN, VXLAN

vm-tracer, Description

vMotion, Data Center Networking

vmstat command, Bash

VMware, Data Center Networking, Quick Things to Know

VNI (VXLAN Network Identifier), VXLAN

VOQ (Virtual Output Queuing), Buffers

vPC (Virtual Port Channel), MLAG, Bow Tie MLAG

VRF (Virtual Routing and Forwarding), Configuring eAPI, Virtual Routing and Forwarding-Virtual Routing and Forwarding

vrf definition command, Virtual Routing and Forwarding

VRID (virtual router identifier), VRP

VRP (Virtual Router Redundancy Protocol), VRP-Miscellaneous, VRP Stuff

advertisement interval, Basic Configuration

authentication for, Basic Configuration

backup routers, VRP

configuration, Basic Configuration-Basic Configuration

features, Miscellaneous VRP Stuff

master router, VRP, Basic Configuration-Basic Configuration

preemption, Basic Configuration, Basic Configuration

priorities for groups, Basic Configuration-Basic Configuration

serving multiple IP addresses, Basic Configuration

shutting down groups, Miscellaneous VRRP Stuff

virtual IP (VIP), VRRP, Basic Configuration

virtual routers, VRRP

VRID, VRRP

vrrp group command, Basic Configuration

vrrp shut command, Miscellaneous VRRP Stuff

VTEP (VXLAN Tunnel End-Point), VXLAN, Understanding

VXLAN-Understanding VXLAN, VXLAN with Manual Control-Plane

VXLAN (Virtual eXtensible Local Area Network), VXLAN-
Conclusion

about, VXLAN-VXLAN

configuration, Configuring VXLAN-Configuring VXLAN

overlay network, Spline

understanding, Understanding VXLAN-Understanding VXLAN

with CVX, VXLAN with CVX-VXLAN with CVX

with EVPN, VXLAN with EVPN

with manual control-plane, VXLAN with Manual Control-Plane-
VXLAN with Manual Control-Plane

with MLAG, VXLAN with MLAG-VXLAN with MLAG

vxlan controller-client command, VXLAN with CVX

VXLAN Gateway, VXLAN

VXLAN Interfaces, Understanding VXLAN

VXLAN Network Identifier (VNI), VXLAN

VXLAN Overlay Network, Spline, VXLAN, Understanding VXLAN

VXLAN Segment, Understanding VXLAN

VXLAN Tunnel End-Point (see VTEP)

vxlan udp-port 4789 command, VXLAN with Manual Control-Plane

vxlan vni notation dotted global command, VXLAN with Manual Control-Plane

W

website resources

Arista, Jayshree Ullal

EOS Central, The EOS Extension System

EOS extensions, The EOS Extension System

SQLite, Using Event Monitor

wget command, Aboot, Aboot

white-box switches/switching, Arista Product ASICs, Why EOS Containers?

wr mem command, Booting with ZTP

wri commands, Configuring Event Handlers

write erase command, Booting with ZTP

write memory command, on-startup-config

X

XML, JSON vs., Screen Scraping

XMPP (Extensible Messaging and Presence Protocol), CloudVision

XON/XOFF setting, [Queue Thresholds](#)

Z

zerotouch cancel command, [Cancelling ZTP](#), [Disabling ZTP](#)

zerotouch-config file, [Disabling ZTP](#)

ZTP (Zero-Touch Provisioning), [Zero-Touch Provisioning](#)-[Conclusion](#)

availability, [Zero-Touch Provisioning](#)

booting with, [Booting with ZTP](#)-[Booting with ZTP](#)

canceling, [Cancelling ZTP](#)

configuration, [Booting with ZTP](#)

CVP using, [Network Provisioning](#)

disabling, [Disabling ZTP](#)-[Disabling ZTP](#)

requirements, [ZTP Requirements](#)

About the Author

Gary A. Donahue (GAD) is a working consultant, trainer, and author who has been in the computer industry for over 35 years. Gary has worked as a programmer, mainframe administrator, Technical Assistance Center engineer, network administrator, network architect, and consultant. Gary has worked as the Director of Network Infrastructure for a national consulting company and has been the president of his own New Jersey consulting company: GAD Technology, L.L.C. He currently works at Arista doing all sorts of traveling and training.

Colophon

The animal on the cover of *Arista Warrior* is the African Harrier-Hawk (*Polyboroides typus*). It lives in sub-Saharan Africa and sometimes moves seasonally to West Africa. It prefers forest and woodland environments that have some water nearby and tends to build its home out of sticks in a large gap in trees or in rocky crevices.

The African Harrier-Hawk is a medium-sized bird of prey that grows up to 65 cm in length and has a wingspan of 160 cm. As a young bird, its body color is brown, but as they mature, they become light gray. Other notable physical characteristics of this bird are that the tips of its wings are black with a white stripe in the middle and its belly has thin black and white stripes. It's an effective hunter not only because of its size, but also because of its double-jointed legs, which it uses to take hold of its prey and to climb. It tends to eat small mammals, the eggs of small birds, chicks, reptiles, frogs, and sometimes oil palm fruit.

Typically a silent bird, an adult African Harrier-Hawk has a weak call. It whistles *su-eeeeee-ooo* or *suee-suee*, and when near its nest, it gives off a high-voiced *wheep-wheep-wheep*. The young in the nest, on the other hand, tend to give off a rapid *ki-ki-ki-ki-ki* sound.

The cover illustration is by Karen Montgomery, based on a black and white engraving from Wood's *Animate Creation*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.