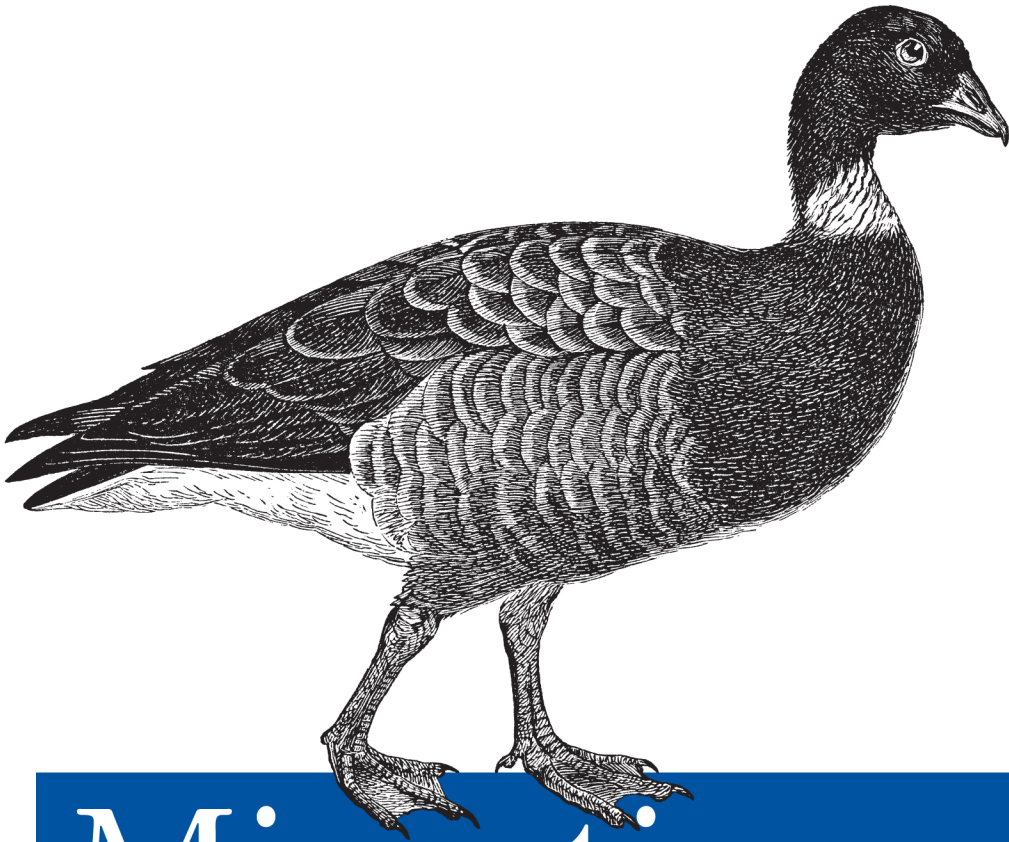


Make Sure IPv6 Doesn't Break Your Applications



Migrating Applications to IPv6

O'REILLY®

Dan York

www.it-ebooks.info

Migrating Applications to IPv6

If IPv6 is to be adopted on a large scale, the applications running on desktop systems, laptops, and even mobile devices need to work just as well with this protocol as they do with IPv4. This concise book takes you beyond the network layer and helps you explore the issues you need to address to successfully migrate your apps to IPv6. It's ideal for application developers, system/network architects, product managers, and others involved in moving your network to IPv6.

- Explore changes you need to make in your application's user interface
- Make sure your application is retrieving correct information from DNS
- Evaluate your app's ability to store and process both IPv6 and IPv4 addresses
- Determine if your app exposes or consumes APIs where there are IP address format dependencies
- Work with the network layer to ensure the transport of messages to and from your app
- Incorporate IPv6 testing into your plans, and use the correct IPv6 addresses in your documentation

Velocity
Web Performance
and Operations

Exceptional web performance and operations are now essential ingredients in every company's online strategy. Learn more about the skills and tools that make this possible at webops.oreilly.com.

Twitter: @oreillymedia
facebook.com/oreilly

US \$24.99

CAN \$28.99

ISBN: 978-1-449-30787-5



O'REILLY[®]
oreilly.com

Migrating Applications to IPv6

Migrating Applications to IPv6

Dan York

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

www.it-ebooks.info

Migrating Applications to IPv6

by Dan York

Copyright © 2011 Dan York. 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://my.safaribooksonline.com>). For more information, contact our corporate/institutional sales department: (800) 998-9938 or corporate@oreilly.com.

Editors: Mike Loukides and Meghan Blanchette

Production Editor: Teresa Elsey

Proofreader: Teresa Elsey

Cover Designer: Karen Montgomery

Interior Designer: David Futato

Illustrator: Robert Romano

Printing History:

June 2011: First Edition.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *Migrating Applications to IPv6*, the image of a brant goose, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-1-449-30787-5

[LSI]

1307712685

Table of Contents

Preface	vii
 1. User Interface Changes	 1
Presentation Format Changes	2
Variable-Length IPv6 Addresses	2
Port Number Colon	3
Classless (CIDR) Notation Versus Subnet Masks	4
Case Sensitivity	5
Validity Checking of Input Fields	6
 2. DNS Changes	 7
Handling AAAA Records	7
Prioritization of AAAA and A Records	8
Happy Eyeballs	9
The Google Chrome Example	9
 3. Application Programming Interfaces (APIs)	 11
Checking the API Transport	12
Checking the API Data	12
An Example: The Session Initiation Protocol (SIP)	13
Handling the API Connection	14
 4. Storage of IP Addresses	 15
Memory Locations	15
Databases	15
Configuration Files	16
Case Sensitivity and Leading Zero Suppression	17
 5. Transport Layer	 19
Dual-Stack versus IPv6-Only	19
Operating System Support	19

Application Dual-Stack Support	20
Multiple IPv6 Addresses	21
Privacy Extensions for IPv6 Addresses	21
Path MTU Discovery	23
Multicast and Broadcast	24
Security	24
NAT and IPv6	24
6. Documentation, Training, and Testing	27
Documentation	27
Training	29
Testing	29
7. Resources and Next Steps	31
Resources	31
Websites	31
Books	32
IPv6 Application Migration Checklist	32
The End? Or the Beginning?	33

Preface

Why This Book?

How *badly* will IPv6 break your application? What do you need to consider to make your application “IPv6-ready”? What questions should you ask?

In the ideal world, your application should “just work” on IPv6, just as it does on IPv4. However, in the *real* world, application issues crop up. These could be as simple as having a user interface field that only allows the input of dotted-decimal IPv4 addresses, or something more fundamental, such as an application binding exclusively to an IPv4 transport interface.

While there have been many books published about IPv6, including O’Reilly’s own *IPv6 Essentials* and *IPv6 Network Administration*, almost all existing books focus on understanding the protocol itself and using it at a network layer. They contain much discussion about using network-level tools and even about creating applications that interact directly with the network. However, the concerns related to IPv6 at the upper application layers are mentioned only briefly, if at all. The Internet Engineering Task Force (IETF) has published [RFC 4038](#), which addresses many of these concerns, but the concerns have not found their way out into mainstream books.

This short book is designed to help you understand what you need to think about to be sure that your app will work as well with IPv6 as it does with IPv4. This book is not so much about all the solutions but rather about the questions you need to be asking.

For IPv6 to truly be adopted on a large scale, ultimately the *applications* running on our desktop systems, laptops, and mobile devices all need to play nice with IPv6. That is the end goal of this book—to help enable individuals, companies, and organizations to migrate their apps to IPv6 so that they can transition their networks into IPv6 networks.

Given that now, in 2011, many companies are just starting to pay attention to IPv6, and given that many apps are just now moving to IPv6, this book will continue to evolve to address issues identified as more applications make the move. I’d love to receive any feedback *you* have on issues you encounter in migrating your apps to IPv6—and I expect that you’ll see updates to this book come out over time.

Is This Book for You?

Are you an application developer? A product manager? A product marketing manager? A documentation author? A training instructor? A system/network architect? This book is designed to help you understand what issues you need to explore with your application.

Developers, you will come away with enough information to go through your application and make the necessary changes. Product managers, you will gain an understanding of what points you need to consider—and what you need to ask of your technical teams. If you are in marketing, documentation, or training, you will get a good sense of what you'll need to think about changing in your materials. And if you are a system/network architect looking at your overall IPv6 implementation, you should leave with a better sense of what changes may need to be considered across the applications that are deployed in your infrastructure.



This book is not a tutorial in the details of IPv6. The focus is on IPv6 issues as they relate to application developers and the book does not get into topics such as network-layer changes between IPv4 and IPv6. If you would like to gain a deeper understanding of IPv6, I recommend also reading [IPv6 Essentials](#) by Silvia Hagen.

What Is in the Book?

To start your dive into IPv6 application migration, [Chapter 1, User Interface Changes](#), explores one of the biggest ways that IPv6 may impact your application: all the many little tweaks you may need to make to your user interfaces. These could be changes to your display or input fields—or something more subtle, like the fact that you may have to think about capitalization in IP addresses.

[Chapter 2, DNS Changes](#), explains the new DNS records for IPv6 addresses and explains how a “happy eyeballs” approach can get users the information they want in the fastest way possible.

[Chapter 3, Application Programming Interfaces \(APIs\)](#), raises questions around APIs, both those your app provides and those your app uses, and how they treat IP addresses.

[Chapter 4, Storage of IP Addresses](#), wraps up this first bit of the book by exploring how you store the IPv6 addresses you receive from user input, DNS, or APIs. If you store them in a memory location, is the location big enough or will there be a buffer overflow? Can a configuration file accommodate both IPv4 and IPv6 address types?

[Chapter 5, Transport Layer](#), drops down briefly into the network layer to discuss issues application developers may need to think about. Does your application work in a dual-stack system? Will it bind to both addresses? Does it need to care about multiple IPv6 addresses or about IPv6 privacy extensions? And is NAT a concern at all?

[Chapter 6, *Documentation, Training, and Testing*](#), explores three areas that complement your actual application—documentation, training, and testing—and asks how you are handling IPv6 in those areas.

Finally, the book wraps up with [Chapter 7, *Resources and Next Steps*](#), providing links to more about migrating applications and a checklist summarizing the key questions from the earlier chapters.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

Constant width bold

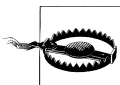
Shows commands or other text that should be typed literally by the user.

Constant width italic

Shows text that should be replaced with user-supplied values or by values determined by context.



This icon signifies a tip, suggestion, or general note.



This icon 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: “*Migrating Applications to IPv6* by Dan York (O’Reilly). Copyright 2011 Dan York, 978-1-449-30787-5.”

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.

Safari® Books Online



Safari Books Online is an on-demand digital library that lets you easily search over 7,500 technology and creative reference books and videos to find the answers you need quickly.

With a subscription, you can read any page and watch any video from our library online. Read books on your cell phone and mobile devices. Access new titles before they are available for print, and get exclusive access to manuscripts in development and post feedback for the authors. Copy and paste code samples, organize your favorites, download chapters, bookmark key sections, create notes, print out pages, and benefit from tons of other time-saving features.

O’Reilly Media has uploaded this book to the Safari Books Online service. To have full digital access to this book and others on similar topics from O’Reilly and other publishers, sign up for free at <http://my.safaribooksonline.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://www.oreilly.com/catalog/9781449307875/>

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>

How to Contact the Author

Dan York also maintains his own site with information about this book, including notes about his IPv6-related presentations, links to webinars, and other resources at:

<http://migratingappstoipv6.com/>

More information about Dan York can be found at:

<http://www.danyork.com/>

Dan can be contacted via email at:

dyork@Lodestar2.com

You can follow Dan on Twitter at:

<http://twitter.com/danyork>

He welcomes feedback about this book through any channel, and would even accept that feedback via pen on paper...although no one seems to do that anymore.

Acknowledgments

First, I'd like to thank Mike Loukides of O'Reilly Media for approaching me about this project. I'd submitted a proposal (which was accepted) to speak on this issue at OSCON 2011 and Mike asked, simply, "Would you like to write a short ebook on the topic?" As the world of publishing is going through such incredible changes, it's great to work with a publisher like O'Reilly that is open to trying out new approaches. I'm looking forward to seeing how this all works out.

My wife once again read through every page and section, offering me critical feedback, even though the subject area has absolutely *zero* interest to her. One of these days, dear, I *will* try my hand at a fiction book or something outside the tech sphere.

I'd like to thank three friends from the IETF and SIP circles—Olle Johansson, Alan Johnston, and Dan Wing—who reviewed my initial outline and provided excellent feedback.

I'd also like to thank my colleagues at my employer, Voxeo, for developing dual-stack versions of our Prophecy and PRISM communications application platforms. It's incredibly awesome when your personal passions and interests (such as IPv6) can intersect so nicely with what is going on at work.

Thanks, too, to all the participants in the IPv6-related webinars I've given recently, who asked me tough questions and deepened my own understanding of the migration challenges with IPv6.

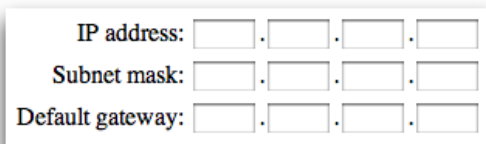
Finally, I'd like to thank all the engineers, administrators, technicians, architects, and everyone else who has been working so long and so hard to bring about the migration to IPv6. Thank you all for continuing to believe and to do all that you do.

User Interface Changes

What is the single biggest way that IPv6 may bite you as an application developer? Sure, if you are dealing with network applications like, say, Wireshark, you're going to be hit by changes down at the socket level. But for the vast majority of applications, which simply need to interact with other applications at a network layer, what is the biggest change?

The *user interface*!

“Huh?” you might be asking yourself...but think about it. How many times have you looked at an application's “Preferences” or “Options” window and seen a screen like this:



IP address: . . .

Subnet mask: . . .

Default gateway: . . .

How well will that work with an IPv6 address?

In fact, as I was writing this book, I happened to notice that the game Minecraft had a new release. [One of the many changes](#) was related to the format of the IPv6 address being entered into the app:

* Fixed IPv6 addresses being parsed wrongly. To connect to a specific port in IPv6, use the format [1234:567::1]:25565

These are just two of the many little ways that IPv6 will mess with your mind and with your app. You'll need to think about and investigate how you present IP addresses for display and how you accept IP addresses for input. This may, surprisingly, be the area that will cause you the most pain and require the most testing. Ideally, we will move away from using IP addresses and toward using DNS domain names, but in some cases,

you may still need to accept or display IP addresses. In this chapter, you'll look at the issues you need to consider.

Presentation Format Changes

The single biggest user-visible change with IPv6 is the move away from the IPv4 “dotted decimal” fixed format of four address blocks, to the IPv6 colon-separated format of varying size. Now instead of an IPv4 address such as:

- 198.51.100.12

you may have a much longer address, such as:

- 2001:db8:1212:3434:1212:5454:1afc:4001

This change raises a number of questions about how you display (output) IP addresses and how you accept (input) IP addresses. Some questions to consider are:

- Are display areas in your application wide enough to accommodate the larger IPv6 addresses? For example, if you have an “IP Address” field in a Preferences window or a pop-up dialog box, will an IPv6 address extend beyond the display area and be cut off or interfere with the design of the window?
- Are your current input fields composed of four boxes for an IPv4 address? Do you need to change them to, for instance, a large text box to accommodate the IPv6 address?

You need to go through your application and look at every instance where you either display or accept IP addresses and determine whether an IPv6 address would work correctly there.

You also need to consider whether you are going to have two *separate* input/output fields for IPv4 and IPv6 addresses. Will you simply have a “IP address” field where a user can enter *either* an IPv4 or IPv6 address? Or will you have different entry fields with another control (such as a radio button) allowing the user to select which address will be entered?

Variable-Length IPv6 Addresses

The second biggest change is the variability in IP address length. With IPv4, addresses have a fixed format of four blocks of decimal numbers. However, in IPv6, the address can vary from the full eight blocks of hexadecimal numbers separated by colons—or blocks with zeros can be compressed with the double-colon (“::”) notation, resulting in a much shorter address. Consider that any of these are perfectly valid IPv6 addresses:

- 2001:db8::1
- 2001:db8:10ff::ae:44f2

- `2001:db8:1212:3434:1212:5454:1afc:4001`

Can your application cope with the variability of the address format?

Now instead of thinking about an IP address as a simple case of four decimal numbers separated by dots, you have to think of it more as a text string to be parsed with separations based on a colon.

Consider these other examples:

- In IPv4, the local loopback address is `127.0.0.1`. In IPv6 it is `::1`.
- IPv4-mapped IPv6 addresses take the form `::FFFF:198:51:100:1`.

Is your application able to handle the variations?

Port Number Colon

Further confusing the matter is the fact that in IPv4 you append the TCP or UDP port number to the IPv4 address using a colon. For example, if you wanted to connect to the web server running on port 8000 of a given system, you would use a URL similar to this:

- `http://198.51.100.22:8000/`

Do you see the problem this is going to cause in IPv6?

If the colon is used as a *delimiter* within the IPv6 address, how do you represent port numbers? One solution is discussed in the next section, but this is a general concern you need to think about. Where in your application are you representing or allowing IPv4 addresses with accompanying port numbers? Can you find all of those instances and ensure that they will also work with IPv6?

The primary way to get around the port number colon issue is to use square brackets around the IPv6 address as specified in [RFC 3986](#). Consider that with IPv4, if you want to connect to a service running on a system, it is very common to simply enter the IPv4 address directly in the URL. For instance, in a web browser, you might enter either of these IP addresses:

- `http://198.51.100.22/`
- `http://198.51.100.22:8000/`

With IPv6, the URLs might then look like:

- `http://[2001:db8:10ff::ae:44f2]/`
- `http://[2001:db8:10ff::ae:44f2]:8000/`

Note that in the second instance the port number is *outside* the square brackets, thus solving the port number colon issue.



This square bracket notation does not just apply to web URLs. It can also be used in the URIs of other protocols. For instance, URIs used with the Session Initiation Protocol (SIP) for Voice over IP might look like this:

- `sip:dan@[2001:db8:34a5::1]`
- `sip:dan@[2001:db8:34a5::1]:5060`

Of course, ideally you are using domain names and only have to type addresses like these for testing purposes.

If your application uses URLs or URIs anywhere, can it parse the square brackets correctly? What will it do if someone enters a URL with an IPv6 address *without* the square brackets?

Classless (CIDR) Notation Versus Subnet Masks

Another change in IPv6 is the way in which you differentiate the *network* portion of the IP address from the *host* portion. With IPv4, if you were giving someone your IP address, you would typically give it to them like this:

- IP address 198.51.100.22 with subnet mask 255.255.255.0

From an application point-of-view, you would typically have two separate fields to enter the IPv4 address and the corresponding subnet.

Now, *some* people in the IPv4 world would use Classless Inter-Domain Routing (CIDR) notation as specified in [RFC 4632](#). You may have also heard this called *slash notation* or something similar. The idea is that instead of providing a specific subnet mask, you specify the *number of bits* that are significant for the network portion of the address. In the example above, the subnet mask of 255.255.255.0 indicates that the first 24 bits (the three blocks of 8 bits) of the address represent the network portion and the remaining 8 bits represent the host portion. This is written in CIDR notation as:

- IP address 198.51.100.22/24

While this notation is certainly used with IPv4 in some areas, many application interfaces still look for a separate IP address and subnet mask.

With IPv6, however, this notation is *mandatory*. All addresses are written using the slash notation. An IPv6 address might be written as:

- IP address 2001:db8:1234:5a:f3a3::22/48

indicating that the first 48 bits (the first three blocks of the address) represent the network portion of the address.

Similarly, if you are given an IPv6 network block, it would typically be written as:

- `2001:db8:1234:56::/64`

indicating that the first 64 bits (or four blocks of the address) represent the network portion.



You may also hear people talk about a network as a “/48” (and saying “a slash 48”) or a “/64” (“slash 64”), as in “You’ll need to get a /48 from your ISP if you are going to have multiple IPv6 subnets.”

From an application developer point-of-view, the main issues here are:

- How do you display the IPv6 address and netmask? If you previously displayed two different fields in a dialog box or command output, you’ll need to merge that information for IPv6 addresses.
- Similarly, how do you accept input for an IPv6 address and the corresponding netmask? Will you have two separate fields? Or will you have users enter it all in one “IP Address” field and parse out the netmask portion?

Case Sensitivity

As your application handles all these different IPv6 address formats, one more subtle change from IPv4 that may impact your app is the fact that IPv6 addresses can be entered with the text in either uppercase or lowercase. Consider that both of these addresses are identical:

`2001:db8:10ff::ae:44f2`

`2001:DB8:10FF::AE:44F2`

And of course someone manually entering an IPv6 address could wind up with some mixture of upper- and lowercase letters. Is your application sensitive to case in any way?

For example, if your app later compares an IPv6 address to past addresses, is the comparison function case-insensitive? Or would it treat the above two addresses as *different* addresses?

A best practice to avoid any case-sensitivity issues is to simply normalize all IPv6 addresses to lowercase when your application receives them as input. This follows the recommendation of [RFC 5952](#) that all IPv6 addresses should be written in lowercase. It does, though, require another step in the processing of an input field that you didn’t have to do with the numeric addresses of IPv4.



[RFC 5952](#) contains a number of other great recommendations with regard to how IPv6 addresses are written. For example, it recommends that the zero compression using “::” be performed on the *longest* string of zeroes in an IPv6 address. While this might seem to be common sense, it is *not* specified by [RFC 4291](#), which defines the architecture of IPv6 addresses. RFC 5952 also provides some other interesting use cases where the textual representation of IPv6 addresses could be a problem, and it is worth a read for anyone working on user interface design for interaction with end users.

Validity Checking of Input Fields

Finally, when thinking about changes in the address format, you need to think about validity testing in any input forms. Typically you want to perform some form of validity testing on any kind of input forms to ensure that a proper IP address has been entered and to guard against security issues such as cross-site scripting, injection attacks, and the like.

With IPv4, this is relatively straightforward. An IPv4 address has a fixed format and uses only decimal numbers. As you’ve seen in this chapter, IPv6 addresses can have a variable format and use hexadecimal numbers. Validity testing *can* certainly be done—it is just a bit more involved.

You also need to go back to the question raised at the beginning of this chapter: are you going to have *one* generic entry field for an “IP address” that accepts both IPv4 and IPv6 addresses? If so, you are going to need to test for both kinds of IP addresses.

Similarly, if you have fields allowing the entry of a URL/URI, you need to consider whether you are checking for IPv6 addresses—with and without the square brackets.

DNS Changes

Ideally you are using domain names instead of IPv6 addresses, but to do so, you need to ensure that your application is retrieving the correct information from the Domain Name System (DNS).

In IPv4, the DNS record pointing a hostname `www` to an IP address is an A record and would look like this in a DNS zone file:

```
www 3600 A 198.51.100.22
```

With IPv6, the new DNS record is the AAAA (also known as a “quad A”) record, as in this example:

```
www 3600 AAAA 2001:db8:12af::53
```

Ultimately, we will get to a point where DNS zone files contain *both* an A and an AAAA record for a host name. For example:

```
www 3600 A 198.51.100.22
www 3600 AAAA 2001:db8:12af::53
```

Today, some operational issues (see below) have prevented widespread deployment of both A and AAAA records for the same host name, but this is the direction in which DNS records *will* go.

Handling AAAA Records

The first question is obviously whether your application can correctly handle AAAA records. When you issue a request for a hostname, does the underlying library even return AAAA records?

If your app does get an AAAA record back, what does it do with the AAAA record? Does it use the record to correctly obtain the IPv6 address for the hostname? Or does your app ignore the returned AAAA record and only look for A records?



It is incredibly important to realize that **AAAA** records *can be retrieved over IPv4*. In other words, just because you can *retrieve* a DNS **AAAA** record does not mean you can *connect* to that site using IPv6.

You can try this yourself if you have only IPv4. Simply go to a command line where you have access to the “**dig**” command and type:

```
dig AAAA ipv6.google.com
```

You will receive back the **AAAA** record for that address. However, whether or not you can connect to that site using the **AAAA** address will depend on whether you have a working IPv6 connection. The point is that your application should not use merely the *availability* of an **AAAA** address as a deciding factor in attempting to connect over IPv6.

Prioritization of AAAA and A Records

A larger question is *which address will your application default to using* when contacting the host? The IPv6 address supplied by the **AAAA** record? Or the IPv4 address supplied by the **A** record?

This is a key question and it is one of the factors that has slowed down deployment of **AAAA** records for major domains. It makes sense to use the IPv6 address first, given that IPv4 addresses will eventually fade away, but this may not always make sense. For instance, in some earlier experiments where sites listed both **A** and **AAAA** records for a web server, a web browser on a computer *without* a full IPv6 connection would try to connect to the IPv6 address first. Eventually it would time out and switch to using the IPv4 address, but the timeout took so long that it made for a horrible user experience.

For that reason, some websites have gone to having an IPv6-labeled version of the site. For instance:

- ipv6.google.com
- www.v6.facebook.com

Obviously it is not ideal to have separate IPv4 and IPv6 DNS names, but it provides a workaround until applications are correctly able to handle both record types.



This is a very real issue, particularly for web browsers. During the writing of this book, the IPv6 tunnel from my home office went down at one point and suddenly all my browsing to IPv6-related sites became *painfully* slow, as those sites had both **AAAA** and **A** records for their main domain name. It was so glacially slow, in fact, that I wound up turning IPv6 *off* on one of my computers simply so that I could get to some of the websites I needed to reach.

Happy Eyeballs

A potential solution to the prioritization issue is a proposal within the IETF referred to as the “happy eyeballs” solution. On the face of it, “happy eyeballs” is an extremely simple idea:

When you receive both an A and an AAAA record, try contacting the site using *both* addresses and then *use whichever address responds first*.

If the IPv6 address is available and able to respond quickly, the communication will happen over IPv6. If an IPv6 connection isn’t available, the communication will occur over IPv4. Similarly, if the IPv6 connection is *slower*, perhaps because it is tunneled over IPv4, the application uses the faster IPv4 connection.

With this approach, the user will be happier, as he or she will get information in the fastest way possible—hence the “happy eyeballs” name.

If more applications adopt this approach, we will get to the point where we *can* have both AAAA and A records listed for the same host name.



This “happy eyeballs” idea is *not* just for web browsers. The idea is applicable to *any* application that could connect to either an IPv6 or IPv4 address. For instance, a Voice over IP (VoIP) softphone is another great candidate for a “happy eyeballs” approach of connecting to whichever IPv6 or IPv4 address replies the quickest.

The Google Chrome Example

In May 2011, Google implemented a variation on this “happy eyeballs” approach in the developer builds of Google Chrome and in the public Chrome builds starting with version 11.0.696.71. The Chrome technique is summarized in [the description of the code change](#):

When a hostname has both IPv6 and IPv4 addresses, and the IPv6 address is listed first, we start a timer (300 ms) (deliberately chosen to be different from the backup connect job). If the timer fires, that means the IPv6 connect() hasn’t completed yet, and we start a second socket connect() where we give it the same AddressList, except we move all IPv6 addresses that are in front of the first IPv4 address to the end. That way, we will use the first IPv4 address. We will race these two connect()s and pass the first one to complete to ConnectJob::set_socket().

In this implementation, Google Chrome tweaks the “happy eyeballs” idea to try the IPv6 address first, *but only for 300 ms*. If no IPv6 connection is successfully made in that time, Chrome will initiate an IPv4 connection and then race the two to see which connection will complete first.

This is a great solution because it still gives preference to IPv6 and saves on an extra network query if the response comes back quickly enough for the connection to proceed entirely over IPv6.

What can you do in your applications to ensure that users are not sitting waiting for IPv6 connections to time out before trying to fall back and connect over IPv4?



If you are interested in diving deeper into DNS and IPv6, O'Reilly published a new short book on the topic in May 2011: [*DNS and BIND on IPv6*](#) by Cricket Liu.

Application Programming Interfaces (APIs)

What about the *application programming interfaces* (APIs) you expose for other applications to use to communicate with your application? What about the APIs you consume in your application? How well do they work with IPv6?

Today it seems that every application or service needs to have some type of API for other apps to use. We live in a time of mashups, where applications are frequently built by pulling data from multiple sources, using APIs of some sort. If your application lives “in the cloud,” where other applications may connect to it across the public Internet, you will have certain APIs publicly exposed to which apps connect. If your application resides on an on-premises server on an internal network, it may allow connections across that local network. Even an application installed on a single machine may expose certain APIs that allow connections from other apps running on that same machine. In all of these cases, your app may also be connecting to APIs exposed by other services and systems.

Some of these APIs may be simple web connections where information is exchanged using a data format like JSON or XML. Others may be exposed ports that allow connections using an industry-standard protocol such as the Session Initiation Protocol (SIP) or the eXtensible Messaging and Presence Protocol (XMPP, formerly known as the Jabber protocol). Still other APIs may use custom proprietary protocols.

The questions you have to answer are these:

- Does your application *expose* any APIs that have an IP address format dependency?
- Does your application *consume* any APIs that have an IP address format dependency?

In both cases, the answers depend upon two factors: the transport and the data payload of the API.

Checking the API Transport

The first step for APIs you are exposing is naturally to determine if an external application can connect to your API using IPv6. This sounds like common sense. I mean... after all, if the *application* is accessible over IPv6, shouldn't the *API* be accessible that way, too?

Unfortunately, the answer will depend on *how* you have exposed the API. If it is, for instance, a simple REST-based call over your regular HTTP connection, all you may have to do is be sure that your system has a valid IPv6 connection and, if necessary, the appropriate DNS AAAA record for the IPv6 address—and of course, make sure that your HTTP server is *listening* on your IPv6 address.

However, many apps may make the API connection available through an *additional* web server running on a different port. In that case, is that additional web server *listening* on the IPv6 address as well? If you have two different web servers with two different configuration files/systems, you may need to ensure that both are listening on both the IPv4 and IPv6 addresses. ([Chapter 5](#), on the transport layer, will dive into this topic in greater detail.)

Similarly, applications that are cloud-based may make their APIs available through a different *domain name* or *subdomain*. For instance, an application running at *www.example.com* may make its API available through *api.example.com*. Is that domain accessible via IPv6? Is there a quad-A record available in DNS? Is the application listening on the IPv6 address if it is different from the regular IPv6 address?

Cloud-based applications can, of course, introduce added complexity into a situation because the API may in fact be handled through a separate *server* than where the primary application runs. The API server may or may not be in the same data center or same network as the main servers. Again, this will obviously vary based on how you have deployed your application, but if you do use a separate server, realize that it needs to be treated exactly like your main server in terms of IPv6 access.

For APIs that your application *consumes*, the steps are very similar. Can you reach the remote APIs over IPv6? Do you need to use any special domain names or port numbers?

Checking the API Data

Once you have a verified IPv6 transport for your API connection, you need to look at the actual data that is being exchanged over that API. Are IP addresses exchanged in some manner? If so, is there a restriction on the IP address type that can be sent? If the API uses a simple format like JSON or XML for the exchange of data, it may be a simple matter of scanning through a “normal” data payload to understand if there are any IP address dependencies. You may, of course, want to read through the API reference or documentation (or source code, if it is your own) to see if there are any *optional*

parameters that could potentially be passed in the API payload but might not be part of a typical payload.

On the other hand, if the API uses a larger protocol, you may need to dive into deeper documentation to determine what type of IP address dependencies are out there.

An Example: The Session Initiation Protocol (SIP)

As an example, let's take a look at the Session Initiation Protocol (SIP) used commonly in Voice over IP (VoIP) and Unified Communications (UC) systems. As part of the initiation of a SIP connection, the SIP packets contain a payload of the Session Description Protocol (SDP), defined in [RFC 4566](#). The challenge here is that SDP hardcodes IP addresses directly into the information that is exchanged. For instance, here is how a basic bit of SDP might look:

```
v=0
o=Example-UA 12224 1749 IN IP4 198.51.100.54
s=SIP Call
c=IN IP4 198.51.100.54
t=0 0
m=audio 49170 RTP/AVP 0
```

The important line is the `c=` line that defines a new connection. As you can see, it clearly defines an IPv4 address:

```
c=IN IP4 198.51.100.54
```

Alternatively, it could have defined an IPv6 address:

```
c=IN IP6 2001:db8:42d7::101
```

However, one current limitation with SIP is that it cannot define *both* an IPv4 and an IPv6 address for the same connection. You *can* put two different `c=` lines in the SDP, but this has the result of actually creating *two separate connections*, which is probably not what you are looking to do.

In the case of SIP, the IETF has actually published a separate document, [RFC 6157, *IPv6 Transition in the Session Initiation Protocol \(SIP\)*](#), that explains issues related to IPv6 for SIP. For SDP payloads, [Section 4 of RFC 6157](#) explores this issue, with the recommendation to use Interactive Connectivity Establishment (ICE), defined in [RFC 5245](#), to determine the best addresses to use *before* sending the SDP packets.

If you are using SIP as part of your APIs, part of your migration to supporting IPv6 might also involve adding ICE support to your SIP stack—or whatever other systems may be developed as there is more deployment of SIP usage over IPv6. The point is that understanding what you need to do to migrate your API usage to IPv6 may be a bit more involved, depending on the protocol that is being used.

Handling the API Connection

Once you are sure that you can connect or receive API connections over IPv6 and that the data exchanged correctly handles IPv6, the remaining question is whether the components of your application that deal with API connections can correctly handle IPv6 addresses.

When an IPv6 address is part of an API communication, can all the functions, modules, libraries, and the like correctly deal with that IPv6 address?

Storage of IP Addresses

Once your application accepts an IPv6 address via an input field or an API or retrieves it from DNS, how does your app actually *store* that address?

And, given that we will be working with dual-stack systems for quite some time, can your application handle *both* an IPv4 and an IPv6 address?

Memory Locations

Consider, for instance, memory locations. With IPv4, you have a rigid 32-bit address. Is your application only allocating a fixed size in memory? What happens when you try to insert the 128-bit IPv6 address into that memory location? Will you have a buffer overflow?

Similarly, in reading from a memory location, have you again addressed the fact that IPv6 addresses may have a variable length, depending on whether address compression was used?

In a dual-stack environment, are you allocating memory space for *two* addresses, so that you can store both an IPv4 and an IPv6 address?

Databases

As with memory locations, you need to look at your database tables where you store IP addresses. Is the field you use to store IPv4 addresses big enough to handle the much larger IPv6 addresses? Do you need to modify any components of your application that interact with the database to address the larger size?

In reading or writing to the database, do you again need to compensate for the variable length of IPv6 addresses? With IPv4, you always knew you had four octets that were represented by decimal numbers. If you applied a format to the results retrieved from your database, it was incredibly trivial to do using IPv4. It's a nice, fixed format.

However, with IPv6 the address could be as short as:

`::1`

or as long as:

`2001:db8:3f4d:ccdd:4179:880a:f00d:5511`

Can your routines that read or write to your database handle this variability?

Configuration Files

Similarly, text-based configuration files have the same issues, regardless of whether they are written by the application or manually edited by users. Are the components of your app that read from or write to any configuration files able to work with IPv6 addresses? Can they handle the variable IPv6 address length?

Configuration files also have another issue you need to think about, namely:

Are there hardcoded IP addresses lurking in the configuration file?

Think of the many times you have browsed through an application's config file and seen instances of hardcoded IP addresses. It could be something as basic as a default address, or it could be an IP address inside of a URL. As a simple example, consider a configuration file that references localhost as `127.0.0.1` in IPv4. In IPv6, localhost is `::1`. Do you need to update the configuration file? Can the configuration file reference *both* an IPv4 and an IPv6 value?

As another example, take a look at the [httpd.conf](#) file used by Apache web servers. To indicate that Apache should listen on a specific IPv4 address, a `Listen` directive is included in the file:

```
Listen 198.51.100.55
```

With IPv6, you need to change the `Listen` directive to have the IPv6 address in square brackets:

```
Listen [2001:db8:3145::100]
```

You can, of course, include a port number after either address. For a dual-stack system where you want the Apache server to listen on *both* IPv4 and IPv6 addresses, you need to include both `Listen` directives:

```
Listen 198.51.100.55
Listen [2001:db8:3145::100]
```

Similarly, when configuring a `<VirtualHost>` directive, you might use an IPv4 address as shown here:

```
<VirtualHost 198.51.100.55>
ServerAdmin webadmin@example.com
DocumentRoot /var/www/html
ServerName www.example.com
</VirtualHost>
```

For IPv6, the file would need to be updated with the IPv6 address (note again the square bracket notation):

```
<VirtualHost [2001:db8:3145::100]>  
ServerAdmin webadmin@example.com  
DocumentRoot /var/www/html  
ServerName www.example.com  
</VirtualHost>
```

Again, similar to the `Listen` directive, if you want a `VirtualHost` directive to apply to both an IPv4 and IPv6 address, you need to have multiple `VirtualHost` directives, although both could simply point to the identical information.

How would your application handle storing IP addresses in a configuration file?

- Would someone need to edit your configuration file to change addresses from IPv4 to IPv6?
- How do you address providing *both* an IPv4 and IPv6 address?
- Have you checked all the components of your application that reference the configuration file to see if they can read both styles of IP addresses? And can they deal with having both addresses in a dual-stack environment?

To this last point, you do need to consider that dual-stack systems will undoubtedly be around for quite a long time and you may therefore need to be sure you can store *multiple* IP addresses, whereas in the past you might have been able to store only a single address.

Case Sensitivity and Leading Zero Suppression

Regardless of how you are storing the IPv6 address, remember the case-sensitivity issue raised earlier. Unlike the all-numeric IPv4 addresses, IPv6 addresses can contain text in the form of the letters a–f. However, the [specification for IPv6 addresses](#) does not mandate whether those letters are uppercase or lowercase, and so you must assume that users of your application may provide IPv6 addresses using either case (or even a mixture).

[RFC 5952](#) *recommends* that IPv6 addresses be entered entirely in lowercase and, for the sake of simplicity, you may want to simply have a routine that normalizes all incoming IPv6 addresses to lowercase before they are stored in memory, a database, or configuration files. Of course, given that some of those storage media may be able to be altered through another means (such as a system administrator editing a configuration file), you may want to also have your application normalize all IPv6 addresses to lowercase when *reading* from where the IPv6 address was stored.

Similarly, the IPv6 address specification allows for an address to either include or not include leading zeroes. These two addresses are identical:

```
2001:db8:10ff::ae:2
2001:0db8:10ff::00ae:0002
```

In fact, you could remove the zero compression and show the address in its full form:

```
2001:0db8:10ff:0000:0000:0000:00ae:0002
```

If you store the address in this longer form and then go to compare it later to the same address in a shorter form, your comparison routine needs to take into account these possible variations.

Again RFC 5952 makes some recommendations here and specifies that leading zeroes must be suppressed and the zero compression indicator (“::”) be used whenever possible (except for the case where there is only *one* zero, in which case the compression indicator should not be used). Given all of this, the recommended address for your app to store of the ones shown here would be:

```
2001:db8:10ff::ae:2
```

Your application needs to consider these points when storing the IPv6 address—and also when reading the IPv6 address in from storage. Will you normalize IPv6 addresses to a standard format? Do you *need* to for your application? If you are doing any kind of comparisons, it would definitely make sense. If you are simply connecting to a network address, you may not need to go through this extra work.

Just a wee bit more complicated than IPv4, eh?

Transport Layer

Ultimately, your application needs to bind to the network layer for the actual transport of messages to and from your application. In the ideal world, this is handled transparently for the application by the underlying operating system or the network bindings of the programming language being used. But is it?

Ideally, many of you can simply rely on your underlying operating system and skip this chapter entirely, but for those of you who do need to work at the network layer, read on for some points to consider.

Dual-Stack versus IPv6-Only

Eventually, we'll wind up in a situation where we have "IPv6-only" network stacks and will simply connect out to the network via IPv6. Whether that will occur in our lifetimes, though, is an open question. The reality is that we will be using "dual-stack" systems throughout the "transition" from IPv4 to IPv6 and potentially for quite some time after that. The challenge with dual-stack systems is in verifying that your application can in fact access both IPv4 and IPv6 network stacks—and that your application does choose to bind to both network stacks.



The availability of a dual-stack solution will depend largely upon the target platform on which your application runs. For instance, some embedded systems may not have the processing power or the memory to run multiple network stacks. Your options may then be limited to using either IPv4 or IPv6, but not both.

Operating System Support

As this book is being written in 2011, all current releases of the major operating systems support a dual stack. Microsoft Windows, Mac OS X, and Linux all have native IPv6 support. Whether or not that support is *enabled* on your particular server may be a different question—but the support is in the operating systems.

If you drop to a command line and type either `ifconfig` (Mac OS X, Linux) or `ipconfig` (Windows), you should see that your current network interfaces have IPv6 support, at least for a link-local IPv6 address. If you do not see IPv6 addresses, you may need to enable IPv6 support through the network configuration for your operating system.

Application Dual-Stack Support

Assuming that your system does have IPv6 support, the next question is *does your application need to be changed to bind to the IPv6 address?* Now, for many of you reading this book, your application may be at a high level, where you rely on calls to the underlying language or operating system. But some of you may in fact be writing at a low enough level where you have to care.

Here is a quick example. Consider this extremely basic Node.js application that runs on a server. All it does is wait for incoming HTTP connections; once such a connection is received, the app sends back “Hello World!” and logs a message to the console:

```
var http = require('http');

var handler = function (request, response) {
  response.writeHead(200, {"Content-Type":"text/plain"});
  response.end ("Hello World!\n");
  console.log("Got a connection from " + request.connection().stream.remoteAddress);
};

var server= http.createServer();
server.addListener("request",handler);
server.listen(80,"198.51.100.22");

console.log("Server running on localhost at port 80");
```

The critical part for our purposes is the line that binds our application to an IPv4 address:

```
server.listen(80,"198.51.100.22");
```

The issue here is that this binds *only* to the IPv4 address. If we want to also bind to the IPv6 address, we need to explicitly add a section to the app so that it binds to both the IPv4 and IPv6 addresses:

```
var http = require('http');

var handler = function (request, response) {
  response.writeHead(200, {"Content-Type":"text/plain"});
  response.end ("Hello World!\n");
  console.log("Got a connection from " + request.connection().stream.remoteAddress);
};

var server= http.createServer();
server.addListener("request",handler);
server.listen(80,"198.51.100.22");

var server6= http.createServer();
```

```
server6.addListener("request", handler);
server6.listen(80, "2001:db8:42d7::2");

console.log("Server running on localhost at port 80");
```

Now, again, your application and language of choice may not require you to get down to this level of detail, but you need to investigate to determine what changes you may or may not need to do within the part of your code that handles interaction with the network itself.

Multiple IPv6 Addresses

Another challenge with IPv6 is that each network interface can have *multiple* IPv6 addresses. Right from the start, each interface is going to have a **link-local** IPv6 address by default. Assuming you have real IPv6 connectivity, each interface will also have a **global** IPv6 address (even if “global” actually means within your network). You also can assign additional IPv6 addresses to each network interface.

Which address should your application use?

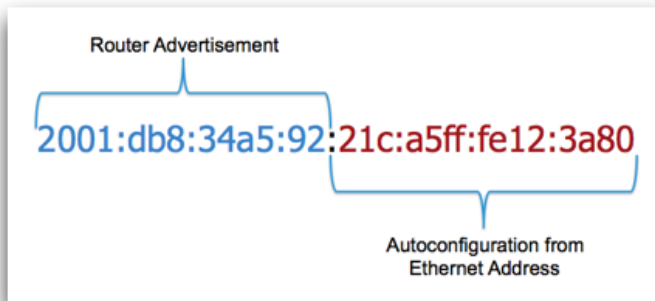
The truth is that you shouldn’t have to worry about this choice. [RFC 3484, Default Address Selection for Internet Protocol version 6 \(IPv6\)](#), defines a selection algorithm that must be implemented by IPv6 network stacks. You should just be able to trust the operating system here...but it’s important to consider if you are trying to debug any connection issues.

Privacy Extensions for IPv6 Addresses

One other fact about IPv6 addresses that you may need to be aware of is that the IPv6 address for a given system may *change* over time if “privacy extensions” are enabled.

To explain what is going on, let’s back up a step and talk about the “autoconfiguration” of IPv6 addresses. As explained in [RFC 4862, IPv6 Stateless Address Autoconfiguration](#), systems on an IPv6 network do not need to use the Dynamic Host Configuration Protocol (DHCP) to receive their IPv6 addresses. Instead, a system can automatically create the IPv6 address by combining a “router advertisement” sent out by the local router with a unique identifier for the network interface. The router advertisement provides the “network” portion of the IPv6 address and the unique identifier for the interface provides the “host” portion of the IPv6 address.

In practice, this “unique identifier” is the IEEE identifier burned into the network interface card at its time of manufacture, commonly known as its “MAC address” or “Ethernet address.” Thus, the creation of an IPv6 address looks like this:



This works wonderfully and provides a very easy way for a system to autoconfigure itself and connect out to the network. There is no need to wait for a DHCP exchange to get on the network. Moving between subnets is also trivial, as the system can simply autoconfigure itself on the new subnet as soon as it gets the router advertisement with the new network prefix.

Now, one problem with this approach is that the *host* portion of the IPv6 address is now *globally unique*, which is a very different concept from IPv4. If you were to go around with your laptop or mobile phone from one IPv6 network to other IPv6 networks (perhaps in that far-off day when cafes all have IPv6 WiFi), you would leave a trail of where you had been, based on the unique host portion of your IP address. An attacker with access to network traces or address logs could do some data mining and correlate your network usage across multiple networks, conceivably learning your location or gaining some insight into the type of traffic you generate and applications you use.

Additionally, an attacker even just watching your IPv6 packet stream go across a network could make guesses as to the type of system you had, based on the autoconfigured host address, and tweak his or her attacks to work with your operating system or device type.



For example, you can go to the IEEE's registry of public Organizationally Unique Identifiers (OUIs) at <http://standards.ieee.org/develop/regauth/oui/public.html> and enter in the first three blocks of an Ethernet address to determine the vendor of that specific network interface card. I can look at my link-local IPv6 address on the computer where I am writing this book and see that it begins with: fe80::21f:5bff:. With a little bit of knowledge about how to interpret that address, I can know that the Ethernet address of my computer begins with "00:1f:5b". I can then go to that website and enter the address as "00-1f-5b" and find out that the OUI was issued to Apple Computer—and indeed, I am writing on an iMac. An attacker with this knowledge could then try to attack my system with attacks against Apple operating systems.

Because of these security and privacy concerns, the IETF came out with [RFC 4941, Privacy Extensions for Stateless Address Autoconfiguration in IPv6](#). If this is enabled on a computer, two things will happen:

1. The autoconfigured host portion of the address will be randomly generated so that it has no relationship to the Ethernet address.
2. A “lifetime” will be assigned to the address, so that at the end of that time period the IPv6 address will be destroyed and recreated using this same randomized process.

The primary issue here for you as an application developer is the temporary nature of these IPv6 addresses if you have some reason to rely on an IPv6 address for an extended period of time. You may need to account for this possibility of changing addresses or focus instead on using domain names that could be dynamically updated by the systems when the IPv6 address is changed.



IPv6 networks are not *required* to use stateless address autoconfiguration. While autoconfiguration is easy and convenient, some organizations may instead opt for the greater control over addressing provided by DHCP and use DHCP for IPv6, defined in [RFC 3315](#). Using DHCP would also get around this entire privacy concern because the IPv6 addresses would be assigned from the DHCP address pool and would not include any sign of the network interface’s Ethernet address.

Path MTU Discovery

Diving again down into the weeds, if you want to send large blocks of data with IPv4, your network stack would send the packets out on the local network using the *maximum transmission unit* (MTU) of the underlying network (for example, Ethernet or PPP). If somewhere between your application and the destination there was a network segment with a *smaller* MTU, the router or other network device connected to that segment would *fragment* individual packets if necessary to fit the smaller MTU. The packets would then need to be reassembled at the destination.

However, in IPv6, such fragmentation *in the network path* is not allowed. Instead, IPv6 uses Path MTU Discovery, defined in [RFC 1981](#), to find the smallest MTU in the path between the source and the destination. Once the Path MTU is found, all packets are sent *from the source* with that MTU. In theory, this would let IPv6 connections make use of underlying network technologies with larger MTU sizes. In practice, however, what this means is that basically all packets are sent on a typical Ethernet network with the minimum MTU of 1280 defined in [Section 5 of RFC 2460](#).

Multicast and Broadcast

Does your application rely on either multicast or broadcast packets in IPv4? If so, you need to be aware that these aspects of IP have fundamentally changed in IPv6. A full discussion of the changes is beyond the scope of this book, but suffice it to say that there is no longer any concept of a “broadcast” packet—and multicasting is baked into the core of IPv6. To learn more about multicast addresses in IPv6, you can read [section 2.7 of RFC 2373](#) and then [RFC 3306](#), which extends IPv6 multicast addressing to allow the dynamic allocation of multicast addresses. O’Reilly’s *IPv6 Essentials* also has a good section on the changes in multicast addressing in IPv6.

Security

At an *application* level, the security aspects of IPv6 are very similar to IPv4. You still will have firewalls. You will still need services to traverse the firewalls. In Linux and UNIX systems, you still have the `iptables` command, although for IPv6 it may be `ip6tables` instead. You still have port numbers, services...at the application layer, IP security is still very much the same as IPv4. (Down at the *network* layer, however, it has changed substantially.)

Two items you should know about, though. First, IPSEC is now mandatory for every IPv6 stack. With IPv4, you often had to install additional software to support IPSEC. With IPv6, IPSEC is baked into the core of the protocol and is mandated to be available. Note that this does not mean that IPSEC is *enabled*—it just means that it has to be *available*. If your application needs transport encryption, IPv6 may make that a bit easier.

Second, while it is incredibly easy to set up an IPv6 tunnel via free services such as [Tunnelbroker.net](#) and put your office or test lab on the IPv6 Internet, please remember that when you use a service like this *you are directly connected to the public Internet!* There is no NAT or firewall in the way: you have a direct connection from whatever machine is the tunnel endpoint out to the public IPv6 Internet. Please make sure that you are running some type of IPv6 firewall on that system to protect your system and network.



For those looking for a deep dive into IPv6 security, the U.S. National Institute of Standards and Technology (NIST) published a great document in December 2010 entitled [Guidelines for the Secure Deployment of IPv6](#).

NAT and IPv6

Finally, a note about Network Address Translation, a.k.a. NAT. We have all gotten used to NAT and its companion, Port Translation, in IPv4 networks. Probably all of

our home networks and the vast majority of business networks use NAT to hide the entire internal network behind a *single* public IPv4 address.

We had to use NAT. There simply weren't enough IPv4 addresses.

And even now, with IPv4 address allocations exhausted, some ISPs are considering so-called Carrier-Grade NAT or Large Scale NAT as a way to hide their entire *network* behind a single public IPv4 address, thus extending even further the lifetime of IPv4 usage.

However, NAT creates numerous problems in terms of network architecture, many of which are enumerated in [RFC 2993](#) and countless other documents and articles online. On a fundamental level, NAT breaks the “end-to-end” principle, where the intelligence resides in the network endpoints and they are able to reach other endpoints by direct addresses. [RFC 4924](#) provides a recent summary of the end-to-end principle. For instance, endpoints that use real-time communications using the SIP protocol are unable to communicate with another endpoint across a NAT device without some kind of assistance in terms of an application-layer gateway (ALG), SIP-aware firewall, or session border controller (SBC). All of these add complexity and challenges to the SIP-based communication.

The *promise* of IPv6 is that with such a large address space there is no longer any need for NAT. NAT can die. This point has been argued quite forcefully within the IETF in many documents, including [RFC 4864](#), [Local Network Protection for IPv6](#). The reasoning is extremely solid and makes a compelling case.

However, the *reality* of IPv6 is that we will still see NAT being deployed with IPv6. Certainly within many enterprises, network architects and administrators have become wed to NAT for a variety of reasons and will continue to use NAT for both IPv4 and IPv6. Instead of [RFC 1918](#) private IPv4 addresses and firewalls with NAT, they will use [RFC 4193](#) Unique Local Addresses (ULAs) for IPv6 and firewalls with application-layer gateways (ALGs). The results will be the same: an internal network using private, non-routable addresses connected out to the global network through a limited number of public addresses.

A July 2010 document from the Internet Architecture Board (IAB) captures the thinking from the two sides quite well: [RFC 5902](#), [IAB Thoughts on IPv6 Network Address Translation](#). It includes this problem statement:

The discussions on the desire for IPv6 NAT can be summarized as follows. Network address translation is viewed as a solution to achieve a number of desired properties for individual networks: avoiding renumbering, facilitating multihoming, making configurations homogeneous, hiding internal network details, and providing simple security.

The document goes on from there to discuss each of these concerns, the architectural issues around NAT, and potential solutions. The document concludes with the understanding that NAT *will* be deployed for IPv6 if adequate solutions are not found for the reasons why NAT is desired.

The religious war between those who see NAT for IPv6 as inherently evil and those who see NAT for IPv6 as incredibly useful will undoubtedly continue for quite a long time.

For you as an application developer, there are a number of key points to consider:

- In an ideal environment, you may not have to worry about NAT with IPv6 and can indeed provide a globally unique IPv6 address to a remote connection. Keep in mind that this doesn't mean the remote connection can immediately connect to your system. Firewalls are still there in IPv6 to prevent unauthorized connections.
- You may find that some networks will implement Unique Local Addresses (ULAs) and have private IPv6 addresses with application-layer gateways to connect to the public Internet. In these cases, you still have techniques like STUN, [RFC 5389](#), and TURN, [RFC 6156](#) for IPv6, that are functionally the same as their IPv4 counterparts to help you connect across a firewall and/or NAT.
- For “offer/answer protocols,” such as SIP for real-time communications, there is Interactive Connectivity Establishment (ICE), defined in [RFC 5245](#), which involves a negotiation session between two endpoints where they agree on which IPv4 or IPv6 addresses to use for communication.
- The [BEHAVE Working Group](#) within the IETF continues to look at NAT issues with a focus on translation between IPv6 and IPv4. Expect to see more documents and recommendations coming out of this group. Note that there is [a document](#) heading toward “Experimental” status outlining how to do NAT in an IPv6-to-IPv6 environment.
- IPv6 is really only now starting to be actually deployed in larger networks. As the deployment grows and actual operational experience accumulates, we'll undoubtedly see which of the various paths are used in terms of NAT usage—or not—within IPv6 networks.

No easy answers, unfortunately.

Documentation, Training, and Testing

While the bulk of this book has been focused on your application itself, three areas that complement your application also need to be considered: documentation, training, and testing.

Documentation

When you write documentation for your application (you *are* writing documentation, aren't you?), what IPv6 addresses are you using in your examples?

If you just use a random IPv6 address, it could turn out to be someone's *real* IPv6 address. Even with the IPv6 address space being so huge, there still is the remote chance that your documentation choice could collide with a real address and potentially cause traffic or routing issues. If you use your *own* IPv6 addresses, well, are you sure you want the traffic of people typing in examples and hitting your systems? (And maybe I'm personally just too concerned about security, but I don't want to have people probing around my systems!)

Thankfully, the good folks at the IETF solved this issue for us with [RFC 3849, IPv6 Address Prefix Reserved for Documentation](#). The magic IPv6 address prefix to use for documentation is:

`2001:db8::/32`

That prefix has been permanently allocated for documentation purposes and will never be assigned to an actual end party. It is, as they say, “nonroutable.”

If you look at the IPv6 addresses I have been using throughout this book, you'll see that they all use the `2001:db8:` prefix, although with various lengths:

```
2001:db8::1
2001:db8:10ff::ae:44f2
2001:db8:1212:3434:1212:5454:1afc:4001
```

When you are writing your documentation, if you need to show communication occurring between multiple IPv6 networks, you can simply subdivide the `2001:db8::` address space into appropriate subnets. For example, if you wanted to show two different networks, you could use address blocks such as these:

```
2001:db8:1::/48 with addresses such as 2001:db8:1::1, 2001:db8:1::2, etc.
2001:db8:2::/48 with addresses such as 2001:db8:2::1, 2001:db8:2::2, etc.
```

Or you could make the address ranges a bit more distinct visually:

```
2001:db8:1::/48
2001:db8:9999::/48
2001:db8:5000::/48
2001:db8:af15::/48
```

The inclusion of letters makes for interesting documentation possibilities. You could simply have ranges like these:

```
2001:db8:aaaa::/48
2001:db8:bbbb::/48
2001:db8:cccc::/48
```

or you could get a bit more creative:

```
2001:db8:feed::/48
2001:db8:fa11::/48
2001:db8:d00d::/48
2001:db8:bad:f00d:/64
```

On this last example, note that while the earlier examples showed a `/48` address range, you can certainly use more specific subnets, such as a `/64`, or even smaller if you need to do so.

Regardless of whether you write about a single or multiple networks, as you write your documentation (or training materials), please use this `2001:db8:` IPv6 prefix for any examples in your documentation. It's definitely the safest way to proceed.



Did you know that there is a similar range of IPv4 addresses set aside for documentation? While not widely used in my own experience, the IPv4 address ranges are defined in [RFC 5737](#) and consist of:

192.0.2.0/24
198.51.100.0/24
203.0.113.0/24

The first 192.0.2.0/24 range was reserved way back in 1989 in [RFC 1116](#), while the second two ranges were added in 2010 so that documentation authors could more easily write about multiple networks without fear of using conflicting addresses. My experience is that authors are more likely to use either the 10.x.x.x/8 or the 192.168.x.x/16 IPv4 address blocks defined in [RFC 1918](#). Regardless of which you choose, the point is that you want to use nonroutable IPv4 addresses in documentation so that there is no conflict if someone attempts to connect to that actual IP address when following along with your documentation.

While on the topic of addresses to use in documentation, are you aware that `example.com`|`net`|`org` have all been set aside as example domain names for you to use in documentation? This is specified in [RFC 2606](#).

Training

Beyond the user guide or similar documents related to your application, do you have *training* materials for your application?

Do you have actual training classes with associated courseware or slides? Online videos or screencasts?

Do any of these training materials need to be updated for IPv6? Do you need new screen captures for your documentation? Do you need to provide guidance on using your application with IPv6?

Odds are that if you have made any changes to address the issues raised in earlier chapters, you will have some work to do here to bring your training materials up to date, even if IPv6 usage or configuration is only a new appendix or additional section.

Testing

As you get a new release ready for your application, have you incorporated IPv6 testing into your test plans? If you have unit tests that are automatically run, have you created appropriate tests that stress the IPv6 interfaces? Do you test out all the IP address entry fields, storage, and the like, to verify that IPv6 addresses can be handled?

If you have a quality assurance (QA) team that tests your app, do their test plans include a segment on IPv6? Do they have an IPv6 test lab where they can actually test your application in a live environment?

Exactly what you need to test will obviously vary depending on your application, but whatever the case, do not forget to include IPv6 testing in your plans.

Resources and Next Steps

So...are you ready to migrate your applications to IPv6 now? At this point, you may either be saying “Sounds like a piece of cake!” and are ready to go—or be saying “Sounds like a nightmare!” and hoping that IPv4 stays around for a *very* long time. Odds are you are somewhere in between. Let’s wrap up this book with some lists and a checklist to get you started in your migration.

Resources

Let’s start with some lists of resources to help you with migrating applications to IPv6.

Websites

The number of IPv6 resource sites is constantly expanding. Rather than include links to specific sites that will surely become outdated, I want to include list a couple of external sites where the list of links can be more easily updated.

<http://migratingappstoipv6.com>

My site for the book, where I will include a list of all the URLs referenced in this book, along with a number of tutorials I’ve given about IPv6 and information about how to get started with IPv6. The site also includes a blog that I will update with new information about migrating applications to IPv6, as well as upcoming events and webinars related to IPv6.

<http://www.oreilly.com/catalog/9781449307875/>

O’Reilly’s site for this book with errata, examples, and more information.

<http://tools.ietf.org/html/rfc4038>

RFC 4038, *Application Aspects of IPv6 Transition*, is an IETF document that provides another approach to the subject of migrating applications to IPv6. The authors go into more detail on some topics than I have, less detail on other topics, and cover a few topics I didn’t (and vice versa). Worth a read to get another view on application migration.

Books

O'Reilly has several excellent books on the topic of IPv6.

IPv6 Essentials

Silvia Hagen's book is probably the premiere book diving into the intricacies of the IPv6 protocol. While it has some information on network administration, it is much more targeted at developers and engineers who need to understand IPv6 at the protocol layer. Outstanding book for anyone looking to really understand IPv6.

IPv6 Network Administration

This book from Niall Richard Murphy and David Malone explores the network administration side of IPv6, with information on configuring IPv6 on various operating systems, routing IPv6, deploying IPv6, running services, and much, much more. The book was published in 2005, and some aspects of the configuration on operating systems have changed a bit, but the fundamentals remain the same.

DNS and BIND on IPv6

The newest book out there (except for the one you are reading), this short book by Cricket Liu builds on her experience with her *DNS and BIND* book to deliver a concise focus on what you need to know about DNS and IPv6.

IPv6 Application Migration Checklist

Here is a simple checklist hitting the main points of the previous chapters:

- ☐ When displaying IP addresses, can your application correctly display the longer IPv6 addresses?
- ☐ When receiving IP addresses as input, do the entry boxes in your app allow for the entry of IPv6 addresses?
- ☐ In the display or input of IPv6 addresses, can your application handle the variable length of IPv6 addresses?
- ☐ Does your app correctly use or accept the “square bracket” notation to allow port numbers to be displayed after an IPv6 address?
- ☐ Does your app correctly handle the input and display of subnet masks using CIDR notation?
- ☐ Is there a need to be concerned about case-sensitivity? Do you need to normalize IPv6 addresses on input to be entirely lowercase?
- ☐ If you perform validity checking on an input field for an IP address, have you updated that validity checking for IPv6?
- ☐ Can your application handle both A and AAAA records from DNS?
- ☐ Does your app implement any kind of “happy eyeballs” mechanism to try to connect over both IPv4 and IPv6?
- ☐ Does your app expose any APIs where there is an IP address format dependency?

- ❑ Does your app consume any APIs where there is an IP address format dependency?
- ❑ Can the components of your application that work with your API correctly handle IPv6 addresses?
- ❑ If you store IP addresses in either memory or a database, is the location or field large enough for an IPv6 address? Or could there be a buffer overflow or database error?
- ❑ For dual-stack systems, can you store two IP addresses in memory, database, or a configuration file for both IPv4 and IPv6?
- ❑ In storing addresses, are you compensating for the variable length, case-insensitivity, and zero compression in IPv6 addresses?
- ❑ Are there hardcoded IP addresses lurking in your configuration files?
- ❑ Can your application work in a dual-stack environment? Or will it only work with either IPv4 or IPv6?
- ❑ If your app runs in a dual-stack environment, does it in fact bind to both interfaces?
- ❑ Does your app need to get down into network layer issues? Does it use multicast or broadcast or have to worry about MTU discovery?
- ❑ Is NAT traversal a concern for your application?
- ❑ Will your documentation materials need to be updated to reflect any changes to the user interface?
- ❑ Will your training materials need any updates?
- ❑ Do your test plans now incorporate IPv6?

The End? Or the Beginning?

Do you have a sense now of how IPv6 will work with your application? Will it break your app? Will your app “just work”? Or will it need a few tweaks and changes? Do you think you understand what questions you need to ask about your application for it to be “IPv6-ready”?

My hope is that you are now better equipped to ensure that your applications can work with IPv6. We’re in a bit of a catch-22 with IPv6: application providers are often reluctant to add in IPv6 support because the underlying network is still all IPv4—while network operators are often reluctant to add IPv6 support to their networks because all the applications only support IPv4!

The reality is that with the exhaustion of IPv4 address allocations and with the continued explosion of new devices we’d like to connect to the Internet, more and more organizations will be looking at IPv6 as a potential solution. You can help with that process by doing what you can to ensure your apps play nice with IPv6.

The other reality is that even though IPv6 has been around now for more than 10 years, only now are we starting to see larger deployments and large-scale interoperability

testing. We're going to learn a *great* amount about how networks and applications work with IPv6 in the months and years ahead. I'm sure it will not all be pretty...there will be broken implementations and broken services, I'm sure. But we'll learn together as we collectively join in this experiment of upgrading the Internet while still using it for everything we want to do.

To that end, as *you* learn about migrating your applications to IPv6, I'd love to get your feedback on what you find to be the biggest obstacles you have to overcome. Are they topics covered in this book? Other topics *not* covered in this book? Any particular tips or tricks that you found? Areas that you wished had deeper coverage?

The beauty of this book being available as an ebook is that it can be updated frequently—and those updates can be shared with those who have purchased the ebook. My intent is definitely to update this book periodically as we collectively gain more experience with migrating applications to actual IPv6 deployments. I'd love your help with that—contact information for both O'Reilly and me directly can be found in the [Preface](#). Please do share your experiences so that others can learn from what you've done. Thanks!

About the Author

Dan York has been writing, speaking, and teaching about online communication technology since the mid-1980s. In 1998 he cofounded the Linux Professional Institute (LPI), today the leading global certification program for Linux professionals, and he later served on the board of directors of Linux International. He's written multiple books on Linux and networking; created some of the first courseware about creating websites, back in the early 1990s; developed open source software in many languages, including Python, Perl, LISP, and most recently Node.js; and worked with more XML variants than he can possibly remember, including DocBook, VoiceXML, and CCXML. His writing can be found at Code.DanYork.com and he is on GitHub as danyork.

Dan serves as the Director of Conversations at Voxeo, heading up the company's communication through new and social media including blogs, video, Twitter, and Facebook, with a focus on creating online content helping developers build applications. Previously, Dan served in Voxeo's office of the CTO, focused on analyzing and evaluating emerging technology, participating in industry standards bodies, and addressing VoIP security issues.

Outside of Voxeo, Dan serves as the chair of the VoIP Security Alliance, is the author of the book *Seven Deadliest Unified Communications Attacks*, and was previously the producer and cohost of the weekly show *Blue Box: The VoIP Security Podcast*.

Colophon

The animal on the cover of *Migrating Applications to IPv6* is a brant goose.

The cover image is from *Riverside Natural History*. The cover font is Adobe ITC Garamond. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSansMonoCondensed.

