Preparing Applications for IPv6

A SOFTWARE DEVELOPERS GUIDE TO WRITING AND MIGRATING NETWORKED APPLICATIONS FOR USE ON IPV6 NETWORKS

VERSION 1

ARIN

American Registry for Internet Numbers

TABLE OF CONTENTS

INTRODUCTION

As IPv4 address space begins to run out, there is building pressure to transition to the more spacious IPv6 protocol. For years network engineers and planners have been upgrading routers, installing new network peers and transits, and tackling address schemes in preparation for IPv6. But on the day they turn it on, will their custom-built applications and commercial-off-the-shelf software be capable of listening on an IPv6 network interface? Will these applications allow for configuration of IPv6? If they persist IP address information, will they have reserved enough space in their storage engines for v6 addresses?

In the push to transition to IPv6 the impact on applications is often overlooked when considering the steps needed to prepare for that transition. This guide discusses the needs of networked software applications when migrating to IPv6 and the juxtaposition of assumptions often made during software development for an IPv4-only Internet.

WHO CAN USE THIS INFORMATION

The overwhelming majority of the books available on IPv6 migration are aimed at helping network engineers migrate their network infrastructure to IPv6.There is dishearteningly little information on IPv6 migration aimed at software developers, and so this guide puts them in its cross-hairs. If you are a software architect, software developer or engineer, or computer programmer, this guide will cover topics in your wheelhouse. After reading it, we hope you will have a good grasp on the changes needed for your software to make a transition to IPv6.

Additionally if you operate software from a vendor and have a good understanding of its features and peripheral workings, then this guide can arm you with information necessary to ask that frank question to your vendor: "Does your software work with IPv6?"

HOW TO USE THIS INFORMATION

If you are a developer of a custom application, this guide can be used as a checklist of areas to investigate in your software's code. As you know your codebase better than anybody else, take a look at each section of this guide and determine if it applies to your software. Does your software act as a REST client? Then you should be aware of the issues involved with placing IPv6 addresses in URLs. Does your software take IP address information from a user? Then you will want to look at the input validation and forms. By using this guide as a checklist, you can spare yourself time-consuming exploratory testing, and you can fix the code you know will break before entering into an expensive QA cycle.

If you are the user of custom or off-the-shelf software, you can use this guide to inform yourself about to the issues your software may encounter with an IPv6 deployment. Does your software store IP addresses in a database? Then you can ask your software vendor if the database is capable of storing IPv6 addresses as well as IPv4 addresses. Feel free to pass this free guide on to your software vendor for their use in determining if the software they have created is suitable for the coming IPv6 world.

STRATEGIES FOR APPS SUPPORTING BOTH IPV4 AND IPV6

When preparing your application to support IPv6 you may face the decision of whether to develop a new IPv6-only application or add IPv6 support to your current application

There are a few drawbacks to offering multiple versions of the same application (e.g. an IPv4-only version and an IPv6-only version). First, users may find it difficult to know which version to use as they won't know which one works before translating the name into IP addresses and trying to establish connections to the IP addresses returned. In addition to user confusion, you will also have to maintain two versions of almost the same source code. To avoid these issues, it is recommended to have hybrid applications supporting both IPv4 and IPv6.

An alternative approach could be to have a "wrapper application" that supports both IPv4 and IPv6 and figures out which protocol version should be used so that it can call the IPv4- or IPv6-only applications as necessary. This application would perform connection establishment (or similar tasks) and pass the opened socket to another application. However, these "wrapper applications" would have to do more than just perform a DNS lookup or determine the literal IP address given, and they will probably become more complex than a hybrid application. Therefore, it's probably a better idea to add IPv6 support to your application and get a hybrid application than to develop a "wrapper application".

You can find further information about scenarios and aspects of application transition to IPv6 in RFC 4038 - Application Aspects of IPv6 Transition.[1]

This RFC describes some scenarios that are worth mentioning. The best-case scenario is that of an IPv4/IPv6 application running on a dual-stack platform. IPv4/IPv6 applications on dual-stack platforms will be able to communicate with other applications, irrespective of the version of the protocol stack or the application in the node.

A not-so-good scenario would be an IPv6-only application running on a dual-stack device. If you substitute all IPv4 API references with IPv6 API references, your application will be IPv6-only and will not work in IPv4-only platforms. Some implementations of dual stack allow IPv6-only applications to interoperate with both IPv4 and IPv6 nodes using IPv4-mapped IPv6 addresses when it's necessary. IPv4-mapped IPv6 addresses consist of a prefix and an IPv4 address: ::FFFF:x.y.z.w represents the IPv4 address x.y.z.w

It is important to note that some systems will disable support for internal IPv4-mapped addresses by default due to security concerns. Enabling or disabling this feature is probably not controlled by software developers, so it is recommended that developers write their applications taking into account both environments.

Another scenario that must be taken into account is when an IPv4/IPv6 application is running in an IPv4-only system. Your application may support both IPv4 and IPv6 but you cannot be 100 percent sure that it will always run in a dual-stack system. It could also run in a node that doesn't support IPv6 yet or in a node that has IPv6 disabled by the user. So it is important to take into consideration the situation in which the user can dynamically enable and disable IPv6 support.

An example of an issue that could arise in this scenario was taken from the RFC 4038 - Application Aspects of IPv6 Transition[2], shown below:

> An example is an application that issues a socket() command, first trying AF_INET6 and then AF_INET. However, if the kernel does not have IPv6 support, the call will result in an EPROTONOSUPPORT or EAFNOSUPPORT error. Typically, errors like these lead to exiting the socket loop, and AF_INET will not even be tried. The application will need to handle this case or build the loop so that errors are ignored until the last address family.

1 http://tools.ietf.org/html/rfc4038
2 http://tools.ietf.org/html/rfc4038

SOCKETS

All network applications use sockets of some type and in some manner. Though the trend in application construction has been toward the re-use of existing application protocols, primarily the HyperText Transport Protocol (HTTP), and application frameworks that abstract the socket programming away from the application (see the section below), many applications still use sockets directly.

In any case, the necessary changes depend on the programming language being used and the system calls utilized by the libraries of that language. Applications written in Java will need no modification unless they were specifically coded to only work on IPv4. But applications written in C (and probably C++) will require non-trivial, though manageable changes.

Until version 5.14, IPv6 socket support in Perl was not part of the core language library and was very cumbersome. However, conversion of code with modern Perl versions simply requires substituting the use of IO::Socket::INET with IO::Socket::IP.

For applications written in most other languages, the changes fall generally into two categories: 1) hostname lookups, and 2) generalization of the socket calls to accommodate both IPv4 and IPv6. Hostname lookups on most systems can involve more than the Domain Name System (DNS), but communication across the Internet using hostnames is dependent on it (see the section on DNS).

Python programs require modifications that are typical of these two categories. The socket.gethostbyname call must be changed to socket.getaddrinfo for proper resolution of hostnames to IPv6 addresses, socket creation calls using AF_INET must be changed to AF_INET6 (which works with both IPv4 and IPv6), the calls to socket.inet_aton and socket.inet_atop must be changed to socket.inet_pton and socket.inet_ntop respectively. However, for clients using the socket.create_connection function no changes are necessary as it is IP version-agnostic.

Like Python, Ruby programs that have specified AF_INET will need to specify AF_INET6. Clients using the convenience methods to create sockets should require no modification as those methods will create a proper socket type from a hostname lookup that is IPv6 compatible. However, servers using the convenience methods will need to explicitly specify a bind address of "::".

Converting C applications (and C++ applications using the C API) is similar to that of Python applications. The structural differences between the two conversions rest on the fundamental differences between the two languages. Here, Python's dynamic typing saves the need to explicitly change the objects passed between the library and the application, whereas in C the structures allocated for passing between the C library and the application will need to be converted.

PROXY AND APPLICATION SERVERS

For the services of many applications, managing sockets is a well-trodden task delegated to proxy and application server software and frameworks. No socket code needs to be modified. However, there is an often-overlooked task of configuring the server framework to listen on an IPv6 port.

This is often a very simple task. For example, with Ruby's webbrick it is a matter of specifying that the server should listen on an (or any) IPv6 address with a command-line switch ("-b ::" to listen on any IPv4 and IPv6 address). Likewise, Litespeed needs its binding address set to "[ANY]" instead of "ANY", and Apache's HTTPd should be directed to listen for IPv6 with either the Listen directive or in the Virtual Host configuration (depending on configuration needs).

Beyond configuration, some servers and frameworks require that you verify IPv6 support has been specified when the software was compiled. Some examples of this are PHP and nginx. The snippet below shows the compile options for nginx, among them --with-ipv6:

```
$ nginx -V
nginx version: nginx/1.6.0
built by clang 5.1 (clang-503.0.40) (based on LLVM 3.4svn)
TLS SNI support enabled
configure arguments: --prefix=/usr/local/Cellar/nginx/1.6.0_1 --with-
http_ssl_module --with-pcre --with-ipv6 --sbin-path=/usr/local/Cellar/
nginx/1.6.0_1/bin/nginx --with-cc-opt='-I/usr/local/Cellar/pcre/8.35/
include -I/usr/local/Cellar/openssl/1.0.1g/include' ...
```

For JVM-based servers, the situation is slightly different. The runtime property "java.net.preferIPv4Stack" determines if the sockets in the JVM will listen on an IPv6 address. While the default for this value is false (the JVM will listen on an IPv6 address), many application servers such as JBoss AS / WildFly set the value to true in their startup scripts.

Overall, when moving a networked application to IPv6 it is important to understand what might need to change with an application or proxy server. In some cases it simply requires a configuration change and in others it may require recompilation of the source modules.

FORMAT AND COMPARISON

When represented textually as a string of characters, IPv4 addresses are expressed canonically for most purposes. Leading zeros are the only issue (e.g. 192.168.000.001 and 192.168.0.1 are the same address), and for the most part this is not a concern.

However, IPv6 addresses can be expressed in three formats, none of which obviate the leading zero problem (that is, each format may or may not have leading zeros). The three formats are full, compressed, and IPv4-embedded. Wait, it gets better! An IPv6 address can be compressed and also have an embedded IPv4 address, and the hexadecimal characters can be either in upper or lower case.

The multiple permutations in which an IPv6 address can be represented textually make simple string comparisons impossible. While dealing with the casing is trivial, the other variances cannot be so easily canonicalized. Therefore software executing string comparisons for IP addresses will likely work for IPv4 but will almost assuredly break with IPv6.

REGULAR EXPRESSIONS

One common practice to determine if a string is an IPv4 address or to search for an IPv4 address is to use regular expressions. The regular expression for an IPv4 address is simple to write.

This is not the case for a regular expression to match an IPv6 address for the reasons given above. Creating one regular expression to match the patterns of an IPv6 address is possible, but the result is an unwieldy and lengthy pattern that does not perform very well.

One solution to this problem is to create four separate regular expressions: one for a full-length IPv6 address, one for an address with compression, one for an address with an embedded IPv4 address, and one for a combination of an address with compression and embedded IPv4 address. Depending on your needs, this may work[3].

However, some uses of regular expressions to find IP addresses have no good solutions with IPv6. One such example is the use of the grep command, which may be found within shell scripts.

NUMERIC EXPRESSIONS

An optimization to represent an address as an unsigned 32-bit integer is used for both storage and comparison of IPv4 addresses on most platforms. This technique doesn't work as well for IPv6 because many platforms do not have a 128-bit integer data type (some languages do offer libraries to support integers longer than 64 bits, such as Java's BigInteger and Ruby's Bignum).

3 A search of the Internet will turn up many examples of regular expressions for IPv6. One such example, along with some test code, is here: https://gist.github.com/syzdek/6086792

The following code snippet shows Python natively handling an integer large enough to support an IPv6 address.

```
Python 3.3.2 (default, Nov  7 2013, 10:01:05)
[GCC 4.8.1 20130814 (Red Hat 4.8.1-6)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> n = 340282366920938463463374607431768211456 # is 2^128
>>> n.bit_length()
129
```

Languages such as Python natively handle integers large enough to store IPv6 addresses. And while Ruby makes a distinction between Fixnum and Bignum object types, it will handle the conversion seamlessly. The same is not true for Java, which has an autoboxing feature for conversion of primitive int's to Integer, but it does not autobox for BigInteger. In which case, if you code to use BigInteger where int was previously used, you will need to do some refactoring.

The following code snippet shows how Ruby easily handles the conversion from Fixnum to Bignum.

```
irb(main):001:0> RUBY_VERSION
=> "2.0.0"
irb(main):002:0> n = 340282366920938463463374607431768211456
=> 340282366920938463463374607431768211456
irb(main):003:0> n.class
=> Bignum
irb(main):004:0> i = 1
=> 1
irb(main):005:0> i.class
=> Fixnum
irb(main):006:0> b = i + n
=> 340282366920938463463374607431768211457
irb(main):007:0> b.class
=> Bignum
```

For situations where integers can be no greater than 64 bits, there are simple solutions depending on the needs of the networked application and what it is using the IP address to do. While IPv6 is specified with 128-bit addresses, current allocation policy has the lower 64 bits assigned to the end-site and the upper 64 bits of the address are used to route Internet packets to and from the end-site. In other words, the lower 64 bits are for local use on an end-users network. For example, a residence might be assigned 2001:500:F0:1/64. While the refrigerator at that residence might get an IP address of 2001:500:F0:1:A:B:C:DF, the portion of the address that separates that residence from other residences on the Internet is 2001:500:F0:1 (the upper 64 bits). Depending on how the IP address is used, the application may drop part of the address. For instance, if the application needs to differentiate one end-site from another then only the upper 64-bits are of importance.

In some situations the problem deepens. While a 64-bit numeric value may be available for use, that value is defined as a floating-point data type in which not all of the bits are available for the expression of an integer. This is sometimes seen in Geographic Information Systems (GIS), which are quite frequently used to model IP networks.

Most 64-bit, floating-point data types offer 57 bits of precision (the bits needed to consistently express the integer portion of the floating point value). If an application can tolerate the loss of the lower 64 bits of the IPv6 address, there is another "hack" that would allow it to store the rest of the address in 57 bits.

This hack works by examining the current IPv6 allocations issued by the IANA. As the IPv6 space is so vast, it is likely that the IANA will not make another IPv6 allocation to an RIR for a very long time. Therefore it can be reasoned that of all the IPv6 addresses an application is likely to encounter on the Internet, it will only see the following patterns for the top 16 bits: 2001, 2002, 2003, 2400, 2600, 2610, 2620, 2800, 2A00 and 2C00. That is only ten bit patterns, which could be enumerated into a four-bit field thus reducing the top 16 bits to only four bits. Reducing the upper most 64 bits of the address by 16 bits and then adding back the four-bit mapping field yields a value of 52 bits, which is big enough to fit into the 57 bits of precision of a 64-bit, floating-point data type.

Prefix	Designation	Top 16 Bits	Mapped 4 Bits
2001:0200::/23	APNIC	2001	0
2001:0400::/23	ARIN	2001	0
2001:0600::/23	RIPE NCC	2001	0
2001:0800::/23	RIPE NCC	2001	0
2001:0a00::/23	RIPE NCC	2001	0
2001:0c00::/23	APNIC	2001	0
2001:0e00::/23	APNIC	2001	0
2001:1200::/23	LACNIC	2001	0
2001:1400::/23	RIPE NCC	2001	0
2001:1600::/23	RIPE NCC	2001	0
2001:1800::/23	ARIN	2001	0
2001:1a00::/23	RIPE NCC	2001	0
2001:1c00::/22	RIPE NCC	2001	0
2001:2000::/20	RIPE NCC	2001	0
2001:3000::/21	RIPE NCC	2001	0
2001:3800::/22	RIPE NCC	2001	0
2001:3c00::/22	IANA	2001	0
2001:4000::/23	RIPE NCC	2001	0
2001:4200::/23	AFRINIC	2001	0
2001:4400::/23	APNIC	2001	0
2001:4600::/23	RIPE NCC	2001	0
2001:4800::/23	ARIN	2001	0
2001:4a00::/23	RIPE NCC	2001	0
2001:4c00::/23	RIPE NCC	2001	0
2001:5000::/20	RIPE NCC	2001	0
2001:8000::/19	APNIC	2001	0
2001:a000::/20	APNIC	2001	0
2001:b000::/20	APNIC	2001	0
2002:0000::/16	6to4	2002	1
2003:0000::/18	RIPE NCC	2003	2
2400:0000::/12	APNIC	2400	3
2600:0000::/12	ARIN	2600	4
2610:0000::/23	ARIN	2610	5
2620:0000::/23	ARIN	2620	6
2800:0000::/12	LACNIC	2800	7
2a00:0000::/12	RIPE NCC	2a00	8
2c00:0000::/12	AFRINIC	2c00	9

BEST COMMON PRACTICES

The advice provided regarding regular expressions and numeric representations of IPv6 addresses is intended to help you determine if your software has areas in need of improvement and provide simple advice should you wish to continue using those methods. However, best common practice around the representation and comparison of IP addresses is to use a library tailored for that need. This is especially true now that IPv6 addresses must be considered. Further, use of these libraries typically solves more than just representation issue and simple comparison issues; many handle dealing with multiple presentation formats, subnetting, etc…

The library to use and the changes necessary will depend upon the language and library of the application. Some applications may require no changes at all, such as those written in Ruby, which has had excellent IPv6 support built into its IPAddr class since at least version 1.86 (as of this writing, 2.1 is the current version).

Application uses of Python prior to Python 3.3 can utilize IPy[4]. And applications using Python 3.3 and above can use either IPy or the built-in ipaddress library. Similarly, applications written in Perl have a choice between Net::IP[5] and NetAddr::IP[6].

For applications written in PHP and Java, there is a mixed blessing with the use of their libraries. PHP has long had the benefit of PEAR's Net_IPv4 class, but the IPv6 support resides in the separate class, Net_IPv6[7], which necessitates application logic changes for conditionals between the two types.

Java's built-in InetAddress classes provide a seamless transition to IPv6 and adequate utilities for IP address management, but it does have one glaring issue. Handing any string that is not an IP address to the InetAddress constructors will result in a DNS lookup (and consequently a thrown exception if the DNS cannot answer the lookup). Therefore an application cannot use the classes to determine if a given string is a valid IP address without incurring a penalty; IP address validation must take place prior to using these classes.

DISPLAY AND USER INPUT

It was previously noted that the best common practice for dealing with IP addresses is to use an IP address library to solve the problem of representation of the IP addresses; these libraries also resolve issues surrounding textual presentation and textual validation that are important with user interfaces.

Due to issues with the various textual representations of IPv6 addresses, the IETF has issued guidance regarding the presentation of IPv6 addresses in text form[8]. The IETF's guidance can be summarized into three categories: 1) do not display leading zeros, 2) use the :: notation when possible but only on the last series of zero bytes and never for just one, and 3) lowercase the hexadecimal characters.

Despite the standardization on the production of IPv6 addresses with text, there are other display issues that must also be addressed. As Microsoft notes to their developers[9], IPv6 addresses are unlike IPv4 addresses in

4 IPy can be found at https://pypi.python.org/pypi/IPy/

5 Information regarding Net::IP can be found at https://metacpan.org/pod/Net::IP

6 Information regarding NetAddr::IP can be found at https://metacpan.org/pod/NetAddr::IP

7 More information on PEAR's Net_IPv6 can be found here: http://pear.php.net/package/Net_IPv6

8 RFC 5952 outlines some rules for the display of IPv6 addresses. It can be found here: http://tools.ietf.org/html/rfc5952

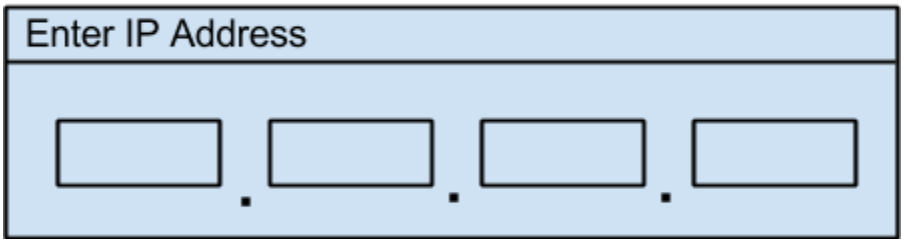
9 Microsoft has issued general advice regarding IPv6 and user interfaces. It can be found here: http://msdn.microsoft.com/en-us/library/windows/desktop/ms740585(v=vs.85).aspx

that their length is unpredictable. This has ramifications for nearly every type of user interface. Where a fixed space was previously allocated in a user interface for IPv4 addresses, that fixed space would necessarily not be sufficient for IPv6. Depending on the nature of the UI layout, the consequences of this can range from the inability to correctly display the entire address to completely ruining the layout of every component.

While there are no universal solutions, some creative thinking about the context of the user interaction may lead to UIs more suited to IPv6. Take for example the following IPv6 address: 2001:db8:7001:5192:6594:9d0a:b09a:564b. There is no way to shorten this address, and displaying it to a user in a dialog with other information is likely to result in the user ignoring the message because it is too much information to readily digest.

One solution may be to uses ellipsis (…) to hide the non-essential parts of the address along with a method that allows the user to see the entire address, like hover text or a pop-up window. Given the example above if the importance of the user interaction is that user know which network is in use, then displaying "… : 13c7 : 7001 : 5192 : …" communicates to the user just the network relevant portion of the address. Likewise, if the importance to the user is their specific host address, "… : 564b" conveys that information. Also note the use of whitespace to break-up the address, which makes the reading of a series of hexadecimal digits easier.

User input of IP addresses also has its own set of problems. It is common to see user interface dialog boxes for IPv4 addresses that have the user enter each set of octets into a separate text box.



Truthfully, such user interfaces are bad design even for an IPv4 world. IPv6 simply makes it much more apparent. It is better to offer one text area for input and take advantage of any facilities allowing users to cut and paste an IP address or to refer to them. Attempting to have users enter a series of characters is a frustrating user experience (again, even for IPv4 addresses).

PERSISTENCE AND DATABASES

Long-term storage of IP addresses, particularly collections of IP addresses, is a fairly common practice with networked applications, especially websites and server software. This can take many forms: logging of the source IP addresses in log files for later analysis (usually IP host to domain name mapping), tracking user logins, and the creation of whitelists (many on-line surveys limit submissions based on subnets, etc…).

No matter the type of storage mechanism, the most immediate concern with the persistence of IPv6 addresses is the amount of storage space needed. While the total amount of extra storage needed for the longer IPv6 addresses can be an issue in constrained environments (e.g. embedded flash memory stores), generally this issue is about the fixed size of fields of a record allocated for IP addresses. While not limited to relational databases, it is a common practice in a relational database to allocate a fixed width column.

Where a column in a database may have been defined as VARCHAR(15) for IPv4 addresses represented as strings, it would now need to be defined as VARCHAR(45). Similarly, if the IP address is represented in binary form, where four bytes of storage was adequate for IPv4 now 16 bytes is necessary.

Like internal storage of IP addresses where a 32-bit integer data type was previously adequate for IPv4-only, the data type for a database column may need some thought depending on your database vendor.

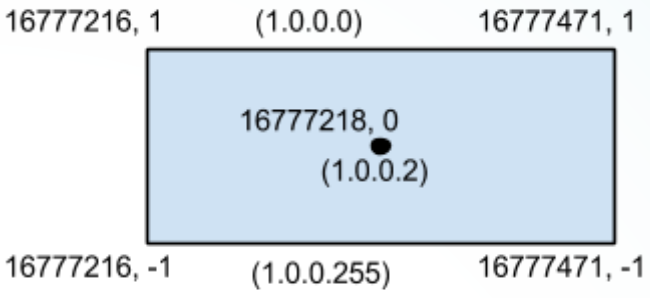
MYSQL

When storing IP addresses as a binary number instead of as a textual dotted-quad string of characters, MySQL's INT data type is perfectly adequate for IPv4. But migrating a column to BIGINT may not be an adequate solution for IPV6 as this data type only supports eight bytes of storage, not the 16 bytes necessary for a full IPv6 address. However, BIGINT is workable if your application can tolerate dropping the lower eight bytes (64 bits) of the IPv6 address.

For many applications, simply storing IP addresses is not enough. The database must also be leveraged to search for IP addresses. A common scenario is determining if an IP address falls within a certain range of addresses for use in whitelisting. Unfortunately, the B-Tree indexes used for standard INT, BIGINT, and VARCHAR columns does not perform very well in these situations.

A trick often used to solve this problem in MySQL is to use the freely available MySQL Spatial extensions. This allows you to model IP networks as rectangles and take advantage of the R-Tree indexes available to polygons. In this situation, the R-Tree indexes perform far better than the B-Tree indexes.

To use this trick, an IP network is modeled as a rectangle where the start and end addresses are represented as integers on the X axis and the points of the rectangle using -1 and 1 for the Y axis. For example, the network 1.0.0.0 to 1.0.0.255 would be a rectangle from 16777216, -1 to 16777216, 1 to 16777471, 1 to 16777471, -1 and back to 16777216, -1. To determine if the IP address 1.0.0.2 is in this network, the spatial extensions are queried to determine if the given rectangle contain the point 16777218, 0.



This all works well for IPv4, but IPv6 integers are much larger and cannot be represented by the data type used by the spatial extensions. Even if your application could tolerate dropping the lower 64 bits of the IPv6 address, there is still not enough storage as the points in the rectangle are represented by 64-bit, floating-point numbers (not fixed point) with only 57 bits of precision.

In this case a possible fix is to map the top 16 bits to a four-bit field (as described above in the section on numeric expressions) in addition to the lowest 64 bits. The reasoning behind this is that all the routable addresses allocated by the IANA have a limited set of high order bit patterns, and that there will be no further allocations by the IANA in the near future. Of course, it was also once thought that the Internet would never run out of IPv4 addresses.

Finally, there is one last consideration with MySQL. If your application makes use of MySQL's INET_ATON and INET_NTOA functions, they will need to be converted to INET6_ATON and INET6_NTOA functions. Both of these new functions work equally well with IPv4 addresses and IPv6 addresses, however unlike the IPv4-only functions they return a VARBINARY data type (instead of an integer).

POSTGRESQL

If the relational database you use is PostgreSQL, there may be little work necessary for IPv6 migration. Since PostgreSQL 7.4 (released in 2003), it has had the built-in datatypes inet and cidr, both capable of representing IPv6 addresses. If your use of PostgreSQL does not use these data types to represent IP addresses, your efforts are just as well spent migrating those columns to inet or cidr instead of resorting to the other solutions necessary with other databases.

Like MySQL, it is also possible to leverage R-Tree indexes to achieve extremely performant queries of IP network ranges. But unlike MySQL, this is accomplished using a purpose built extension called IP4r, which leverages PostgreSQL's GiST extensible indexing feature. Do not let the IP4r name fool you; as of version 2.0 it fully supports IPv6 addresses.

ORACLE AND OTHER DATABASE VENDORS

Unfortunately, the options for storing IPv6 addresses in an Oracle database are limited. Whereas using NUMBER(10) may have been sufficient for IPv4 addresses, the Oracle NUMBER data type is completely unusable for IPv6 as it can represent at most 38 significant digits and the IPv6 addressing scheme requires 39 significant digits. Similarly to MySQL, Oracle has GIS extensions which can be leveraged to use an R-Tree index for fast lookups of an IP address in a given set of IP networks. However, the same problem applies here as it does with regular Oracle tables because the Oracle GIS extensions use NUMBER as the base type for modeling points and polygons.

The solution for storing IPv6 addresses in Oracle falls into two areas: 1) store them as a string of characters, or 2) drop some of the lower-order bits of the address, such as the last 64 bits, and enable the high-order 64 bits to be represented by NUMBER.

As for other databases from other vendors, the answer is similar. You will have to consult your vendor's documentation to determine the correct path forward for your software: either widening a character string column to 45 characters or dropping lower-order bits of the IPv6 address. Though welcomed, it would be surprising if the database software supported an integer type capable of 16 bytes of storage or a fixed-point type capable of representing 39 significant digits.

URLs (which are a kind of URI) are often used by network applications. For example, many mobile apps use URLs for RESTful communication to server-based applications, and those server-based applications might use URLs to communicate with LDAP services (for centralized authentication), databases, or other RESTful services.

The basic syntax of all URLs is as follows (note, it is actually more complicated than being expressed, see RFC 3986):

[scheme]://[host]:[port]/[path][query]

Thus an HTTP URI might be:

https://rdap.arin.net:443/rest/net?ip=198.51.100.1

IPV6 ADDRESSES AS URL HOSTNAMES

As IPv6 addresses are textually expressed with colons (':') and the host and port are separated by a colon, placement of IPv6 addresses in URLs requires that they be escaped by square brackets ('[' and ']'). The following is the correct way to formulate an HTTP URL with an IPv6 address:

https://[2001:500:4:13::125]:443/

For software constructing URLs from its constituent parts, care may need to be taken with how IP addresses are used. Some software, as is demonstrated with the Java code below, may not require any modifications.

```
import java.net.URL;
public class URLEx
{
 public static void main( String[] args ) throws Exception
 {
  URL url = new URL( "https","2001:500:4:13::125","" );
  System.out.println( url.toString() );
 }
}

$ java URLEx
https://[2001:500:4:13::125]
```

Unfortunately, not all URL and URI libraries seamlessly handle the escaping of IPv6 literals. While Java's URL class does the write thing, its more generic URI class does not. Also take for example this Ruby code:

```
require 'uri'
uri = URI::HTTP.new( 'https', nil, '2001:500:4:13::125',
 nil, nil, nil, nil, nil, nil )
uri2 = URI( uri.to_s )
```

```
$ ruby uri_ex.rb
/System/Library/Frameworks/Ruby.framework/Versions/2.0/usr/lib/ruby/2.0.0/uri/generic.rb:214:in `initialize': the scheme https does not accept registry part: 2001:500:4:13::125 (or bad hostname?) (URI::InvalidURIError)
        from /System/Library/Frameworks/Ruby.framework/Versions/2.0/usr/lib/ruby/2.0.0/uri/http.rb:84:in `initialize'
        from /System/Library/Frameworks/Ruby.framework/Versions/2.0/usr/lib/ruby/2.0.0/uri/common.rb:214:in `new'
        from /System/Library/Frameworks/Ruby.framework/Versions/2.0/usr/lib/ruby/2.0.0/uri/common.rb:214:in `parse'
        from /System/Library/Frameworks/Ruby.framework/Versions/2.0/usr/lib/ruby/2.0.0/uri/common.rb:747:in `parse'
        from /System/Library/Frameworks/Ruby.framework/Versions/2.0/usr/lib/ruby/2.0.0/uri/common.rb:996:in `URI'
        from uri_ex.rb:4:in `<main>'
```

The resulting URL generated by the URI library is http://2001:500:4:13::125, which Ruby's own URI libraries cannot use to create a connection. But at least here the code results in an obvious error. The same cannot be said for Python's URI libraries:

```
import urllib.parse
import urllib.request
url_parts = ['https','2001:500:4:13::125','','','','']
url_string = urllib.parse.urlunparse( url_parts )
req = urllib.request.Request( url_string )
resp = urllib.request.urlopen( req )
data = resp.read
```

```
$ python3 urlparse_ex.py
Traceback (most recent call last):
  File "/usr/lib64/python3.3/urllib/request.py", line 1248, in do_open
    h.request(req.get_method(), req.selector, req.data, headers)
  File "/usr/lib64/python3.3/http/client.py", line 1061, in request
    self._send_request(method, url, body, headers)
  File "/usr/lib64/python3.3/http/client.py", line 1099, in _send_request
    self.endheaders(body)
  File "/usr/lib64/python3.3/http/client.py", line 1057, in endheaders
    self._send_output(message_body)
  File "/usr/lib64/python3.3/http/client.py", line 902, in _send_output
    self.send(msg)
  File "/usr/lib64/python3.3/http/client.py", line 840, in send
    self.connect()
  File "/usr/lib64/python3.3/http/client.py", line 1194, in connect
    self.timeout, self.source_address)
  File "/usr/lib64/python3.3/socket.py", line 417, in create_connection
    for res in getaddrinfo(host, port, 0, SOCK_STREAM):
socket.gaierror: [Errno -2] Name or service not known

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "urlparse_ex.py", line 6, in <module>
    resp = urllib.request.urlopen( req )
  File "/usr/lib64/python3.3/urllib/request.py", line 156, in urlopen
    return opener.open(url, data, timeout)
  File "/usr/lib64/python3.3/urllib/request.py", line 469, in open
    response = self._open(req, data)
  File "/usr/lib64/python3.3/urllib/request.py", line 487, in _open
    '_open', req)
  File "/usr/lib64/python3.3/urllib/request.py", line 447, in _call_chain
    result = func(*args)
  File "/usr/lib64/python3.3/urllib/request.py", line 1283, in https_open
    context=self._context, check_hostname=self._check_hostname)
  File "/usr/lib64/python3.3/urllib/request.py", line 1251, in do_open
    raise URLError(err)
urllib.error.URLError: <urlopen error [Errno -2] Name or service not known>
```

The error given is the same error that would be generated if the URI had been passed a non-existent domain name. Python's URI library does not recognize the IPv6 literal, does not escape it, and then attempts to parse the address which results in a hostname of "2001".

The lesson to be learned is that if your software constructs URIs, IPv6 may cause errors in the host portion of the URI. Where code previously blindly copied the host, be it an IP address or domain name, it may now need to check if the host is an IPv6 address and, if so, escape it.

IP GEOLOCATION

It is quite common for server-based applications to tailor services based on the location of the client, and one such method used is to locate the client geographically using the client's IP address (IP geolocation). In this case, changing of the address type from IPv4 to IPv6 may have some implications.

IP Geolocation works by taking the IP address of the system being geographically located and looking it up in a database. How the database is created and how accurate it is depends on the provider of the database. This basic concept is the same for both IPv4 and IPv6 (and hopefully dispels the myth that IPv6 provides location via its extension headers).

Depending on which IP geolocation database is in use, IPv6 may not be supported. For instance, MaxMind only recently (late 2013) updated their database to seamlessly handle both IPv4 and IPv6. Previous incarnations separated the IPv6 geolocation into a separate system.

For some IP geolocation systems, the API into the database has changed to support IPv6, and for the feed-based systems, which enable customers to locally store the data in a relational database such as MySQL, there are likely schema changes needed (see the section above on persistence and databases).

Finally, there is another common myth about IPv6 and geolocation based on an assumption about mapped IPv4 addresses. As mentioned above in the section on address formats, IPv6 can express IPv4 addresses. However, this type of mapping and expression has no relation to a globally assigned IPv6 address that would be used for communication over the Internet. Therefore, the set of data needed to geolocate IPv6 addresses is not the same as the set of data needed to geolocate IPv4 addresses.
IPv6's address structure does not just provide a much, much larger

address space; but it also has new features and changes to some of the old ones found in IPv4. From an application perspective, this changes the handling of broadcast messages and local host addresses.

LOOPBACK AND LOCALHOST

Some client applications send data or create connections specifically to other processes on the same node using IP sockets. Some server applications have special processing rules, such as authorization policies, that change if the software recognizes the connection is from another process on the same node. This communication happens using an IP address known as the loopback address, usually referred to as "localhost".

In IPv4, the loopback address can be any in the range between 127.0.0.0 and 127.255.255.255, though 127.0.0.1 is almost always the address used. With IPv6 there is only one loopback address, ::1. For clients, the most portable way to address a local process is to resolve the "localhost" hostname instead of using an IP address.

For servers, it is best to make use of IP address libraries if available. The following code shows how Java's abstraction of IPv4 and IPv6 classes into a single class hierarchy make the determination of the localhost easy via a common method call. Python 3.3 and above work similarly.

```
import java.net.InetAddress;
public class LoopbackTest
{
  public static void main( String[] args ) throws Exception
  {
    InetAddress i = InetAddress.getByName("localhost");
    boolean b = i.isLoopbackAddress();
    System.out.println("loopback = " + b );
  }
}

$ java LoopbackTest
loopback = true
```

Unfortunately for situations where a library does not solve the problem, resolving the "localhost" hostname might work but it does have some issues, as the following demonstrates.

Consider this Ruby code running on Mac OS X 10.9.3 (the latest at the time of this writing):

```
irb(main):008:0> Addrinfo.ip('localhost').ipv6_loopback?
=> true
```

Now consider the same code running on RedHat's Linux Fedora 20 (also the latest at the time of this writing):

```
irb(main):048:0> Addrinfo.ip('localhost').ipv6_loopback?
=> false
```

Ruby, as well as Java and Python and other languages, are dependent on the underlying operating system to map "localhost" to a loopback address. The out-of-the-box configuration of Mac OS X maps ::1 to "localhost" but not 127.0.0.1. The out-of-the-box configuration of Fedora 20 maps ::1 to "localhost6", but maps 127.0.0.1 to "localhost".

A better solution is to mimic the behavior of the Java and Python libraries, which does not utilize the "localhost" lookup.

1. Is the IP address an IPv4 address?
    a. If so, is the top byte 127?
    b. If so, it is a loopback address.
2. Is the IP address an IPv6 address?
    a. If so, canonicalize the address either textually or numerically.
    b. Is the address ::1?
    c. If so, it is a loopback address.

It should be noted that the Ruby code above was given for demonstration purposes. The proper way to use the Ruby library in this case is as follows.

```
irb(main):049:0> a = Addrinfo.ip('localhost')
=> #<Addrinfo: 127.0.0.1 (localhost)>
irb(main):050:0> a.ipv6_loopback? || a.ipv4_loopback?
=> true
```

BROADCASTS AND MULTICASTS

In IPv4, it is possible for networked applications to send packets to all nodes on a subnet. Multicasting was later added to IPv4. Multicasting is a method of sending packets to a group of other nodes but not necessarily all nodes, and the nodes can be anywhere on the Internet assuming they are on a network that supports multicasting.

IPv6 does not have broadcasting but instead has an improved form of multicasting. As IPv4 multicasting to nodes on the same subnet is effectively the same as IPv4 broadcasting, a portable means of writing code to work with both IPv4 and IPv6 is to use multicasting for both.

Both IPv4 and IPv6 have multicast addresses reserved for addressing all nodes on a local network, which are 224.0.0.1 and ff02::1 respectively. Unfortunately, there is no portable way to address both IPv4 and IPv6 multicast nodes with a single address, so applications will need a mechanism such as a configuration setting to determine which to use.

If it is desirable to have dual-stack nodes multicast on both IP networks, the data being sent will need to be tagged so receiving applications can determine previous receipt of the data. In other words, a dual-stack node receiving multicasts from both networks will receive the same data twice from a dual-stack node sending multicasts to both networks.

Just as there are different all-local-nodes addresses between IPv4 and IPv6, applications with other multicast needs will have to use the appropriate IPv6 address when porting functionality from IPv4. Developers should consult the IANA IPv6 Multicast Address Registry[10].

EMBEDDED ADDRESSES

As mentioned above in the section on address formats, textual representations of IPv6 addresses may incorporate a "dotted-quad" IPv4 address from the lower 32 bits. These embedded addresses come from two ranges in the IPv6 address space, ::0000/96 and ::ffff:0:0/96. These two spaces are known as IPv4-compatible IPv6 addresses and IPv4-mapped IPv6 addresses respectively. The IPv4-compatible addresses have been deprecated and in practice one should only see IPv4-mapped addresses "in the wild".

Depending on the platform, libraries, and language used, these embedded addresses may be represented in multiple ways, and understanding how they may end up being textualized will avoid surprises with log file formats, IP address comparisons, user interface forms, etc… (as mentioned above, comparison of IP addresses should involve an IP address library for best results).

10 The IANA IPv6 Multicast Address Registry can be found at http://www.iana.org/assignments/ipv6-multicast-addresses/ipv6-multicast-addresses.xhtml

The following Java program run on a Fedora 20 Linux system shows the differences.

```
import java.net.InetAddress;
public class EmbeddedAddress
{
   private static void printAddress( String address ) throws Exception
   {
      InetAddress i = InetAddress.getByName( address );
      System.out.printf( "%20s = %s\n",address, i.getHostAddress() );
   }
   public static void main( String[] args ) throws Exception
   {
      printAddress( "192.168.0.1" );
      printAddress( "::192.168.0.1" );
      printAddress( "::c0a8:1" );
      printAddress( "::ffff:192.168.0.1" );
      printAddress( "::ffff:c0a8:1" );
   }
}
```

```
$ java EmbeddedAddress
       192.168.0.1 = 192.168.0.1
     ::192.168.0.1 = 0:0:0:0:0:0:c0a8:1
         ::c0a8:1 = 0:0:0:0:0:0:c0a8:1
  ::ffff:192.168.0.1 = 192.168.0.1
     ::ffff:c0a8:1 = 192.168.0.1
```

Note that the IPv4-compatible address is textualized as a pure IPv6 address while the IPv4-mapped address and the IPv6 address in the IPv4-mapped address space are textualized as an IPv4 address.

The output on the same Linux platform is different for Python as is demonstrated by the following program.

```
import socket
def print_address( address ):
  print( "%20s = %s" % (address,socket.getaddrinfo( address, 80 )[0][4][0]) )

print_address( '192.168.0.1' )
print_address( '::192.168.0.1' )
print_address( '::c0a8:1' )
print_address( '::ffff:192.168.0.1' )
print_address( '::ffff:c0a8:1' )
```

```
$ python3 embedded_address.py
       192.168.0.1 = 192.168.0.1
     ::192.168.0.1 = ::192.168.0.1
         ::c0a8:1 = ::192.168.0.1
  ::ffff:192.168.0.1 = ::ffff:192.168.0.1
     ::ffff:c0a8:1 = ::ffff:192.168.0.1
```

Here the IPv6 address in the IPv4-compatible address range is textualized as an IPv4-compatible address, and the IPv6 address in the IPv4-mapped address range is textualized as an IPv4-mapped address.

Neither the Java example nor the Python example are incorrect, and both are consistent. But attention should be paid to how the addresses are presented in text form.

INFRASTRUCTURE CONNECTIVITY

Multi-tiered or multi-service applications architectures must consider the connections between clients and servers as well as the connections from server to server, such as from an application server to a database server or billing system. Given the expected scarcity of IPv4 addresses, some companies are opting to use them only for their customer-facing interfaces and transitioning back-end and back-office servers to IPv6. Note that this type of austerity strategy often includes site-to-site connectivity, where for example a database server may have an IPv6 connection between its master node at one site and a backup-node at another site.

There are several considerations when looking at the back-end, server-to-server flow of traffic over IPv6.

ENABLING IPV6 IN THE OPERATING SYSTEM

In general, the most commonly used operating systems have had IPv6 support for a few years now. In some cases the IPv6 support is enabled by default. In some other cases IPv6 is disabled by default, and you need to enable it.

Whether IPv6 is enabled by default or not, it is worth mentioning that it could also be necessary to set up one or more interfaces. In some cases the interfaces could take an IPv6 address automatically (if there is a router in the network sending router advertisements (RAs)). However, if the computer is running an app that will have to interact with other systems, the best option may be to assign IPv6 addresses to interfaces statically.

Enabling and configuring IPv6 in an operating system is highly dependent on the operating system itself, and in the case of Linux it even differs from distribution to distribution.

Windows server operators should take special note of Teredo. It is a platform-independent protocol developed by Microsoft that provides a way for nodes located behind an IPv4 NAT to connect to IPv6 nodes on the Internet. If you're planning to enable IPv6 in your Windows box by using Teredo, you should try to find another way as Teredo has been deprecated and Microsoft is no longer offering Teredo mapping services.

LOCAL FIREWALLS AND ICMP TRAFFIC

Once IPv6 has been enabled in some nodes of a network, IPv6 traffic will start to flow in that network and from/to it. Your network may have two or more levels of filtering. The different nodes may have a local firewall; you may also have a network firewall and you could also have a border firewall at the edge of your network.

As a developer you probably won't be responsible for the network firewall or for the border firewall but you could have to manage rules at the local firewall of the server where your application is running.

Those applications that need to interact with servers or other external services will probably need to be listening on a certain port. In order for things to work properly, traffic to and from the port must be allowed in the internal firewall, in the network firewall, and in the border firewall.

The computer running the application may be running a local firewall. If it is a computer with a Unix based operating system, the local firewall could be for example IPFilter (various Unix OS), ipfw (FreeBSD/Mac OS X), NPF (NetBSD), PF (OpenBSD, and some other BSDs), iptables/ipchains (Linux). On the other hand, if it is a Windows computer, the Windows firewall will probably be enabled.

Apart from enabling IPv6 in the operating system, it is also necessary to pay attention to another protocol: ICMPv6. ICMPv6 is the implementation of the Internet Control Message Protocol (ICMP) for IPv6. ICMPv6 is essential to the functioning of IPv6 and it is very useful for troubleshooting, therefore some ICMPv6 messages need to be allowed.

For troubleshooting, ICMPv6 Echo Request, Echo Reply, Destination Unreachable and Parameter Problem messages should be allowed. In order for the Neighbor Discovery protocol to work properly, multicast ICMPv6 messages (Router Advertisement, Router Solicitation, Neighbor Advertisement, Neighbor Solicitation) should also be allowed. Lastly, as IPv6 fragmentation is not possible in intermediate routers and only the source node can fragment packets, Packet Too Big ICMPv6 messages should also be allowed in order for the source node to detect if a packet should be fragmented.

Some ICMPv6 messages are sent from or to multicast addresses, therefore, traffic from and to multicast addresses should also be unfiltered.

Finally, as an interface can have a link local IPv6 address apart from the global IPv6 address, it's also necessary to allow traffic to link-local addresses.

The general recommendation is that the local firewall should be flexible, allowing all the ICMPv6 traffic and traffic to/from multicast and link-local addresses. ICMPv6 should also be allowed in the local network but it should be filtered in the border firewall.

The filters that should be used in the network firewall and in the border firewall are out of the scope of this document, and usually those firewalls will be managed by the Network Administrator and not by you as a developer. You can find more details regarding the filtering of ICMPv6 messages in the RFC4890 "Recommendations for Filtering ICMPv6 Messages in Firewalls"[11].

11 http://www.ietf.org/rfc/rfc4890.txt

DNS CONSIDERATIONS

If an application needs to connect to some service by name, it will use a DNS name resolver to translate that name into a list of destination addresses. It is worth mentioning that the version of the IP protocol used to carry the DNS queries is independent from the protocol version of the addresses included in the DNS data records. Which means that you can connect through IPv4 to a DNS server in order to query for information related to IPv6 and the other way around.

If you want your application to be accessed by IPv6 or to establish communications with other services using IPv6, you will probably want to add (or request your Systems Administrator to add) the corresponding records in your DNS server with IPv6 information for the name that will be used to access your app.

On the other hand, something that needs to be taken into account is that being able to translate a name server into an IPv6 address doesn't mean that the application running on that server supports IPv6.

DNS RECORDS

In IPv4, the DNS record used to translate a hostname into an IPv4 address is called an A record. In IPv6, as IPv6 addresses are four times longer than IPv4 addresses, the new record used to translate a hostname into an IPv6 address is called the AAAA record (sometimes called a "quad A" record). A single hostname could have both an A record and an AAAA record, therefore that hostname could be translated into an IPv4 address and into an IPv6 address.

If you want your application to be accessed through IPv6 and the clients will use DNS to access it, meaning that they will use a name that will have to be translated into an IP address, you should have AAAA records deployed in your DNS server.

For example, if you already have a DNS A record for your hostname example.com to be translated into the IPv4 address 192.168.1.10, you should add an AAAA record for the same hostname to be translated into an IPv6 address as it is shown below:

example.com. IN A 192.168.1.10
example.com. IN AAAA 2001:db8:1:1::10

DNS LOOKUPS

In addition to being accessed by a hostname, your application may need to access other services using names. It will need to do a lookup in order to translate that hostname into an IP address or it will use a library to do the lookup. Applications that do lookups or that trigger lookups need to ensure they query for both A and AAAA records.

There is an issue related to a client application doing a DNS lookup and asking both for A records and for AAAA records. By only doing the lookup, a client application cannot be certain of the version of the peer application. Simply being able to translate a hostname into an IPv4 address and an IPv6 address doesn't mean that the server application running in the system associated to that hostname, supports IPv6. If the application running in that system doesn't support IPv6, the client application will fail to connect to it.

To address this issue, the client application should request all IP addresses (both IPv4 and IPv6) and try all the records returned from the DNS, in some order, until a working address is found.

For example, the resolution function getaddrinfo() from the POSIX standard API returns a list of all configured IP addresses for a hostname. If you're using this function, you could constrain the query to one protocol family (only IPv4 or only IPv6 addresses). However, the recommendation is that you should obtain all the configured IP addresses, both IPv4 and IPv6.

But what do you do once you have IPv4 and IPv6 addresses? You will need to decide if you want to default to one of the versions of the IP protocol or if you will implement an algorithm that helps you decide which IP address to use to establish a connection.

If you wanted to default to IPv6, you could try to establish a connection using the IPv6 address first, however, if something doesn't work your application will be waiting for a time out before trying to establish a connection using the IPv4 address. This results in a bad user experience because the user will perceive the application as being slow.

In April 2012 a document describing "Happy Eyeballs" was published

as RFC 6555[12]. It describes an algorithm that decides which version of the IP protocol should be used to establish a connection to a host. Most web browsers use this algorithm or some variation of it. However, it is not exclusive for web browsers; it's applicable to any application that could connect to either an IPv4 or an IPv6 address.

Basically, Happy Eyeballs says that if you receive both an A and an AAAA record, you should try contacting the host using both addresses and measure the response time. Then you will establish a connection to that host using the IP address that responded first.

RFC 6556, "Testing Eyeball Happiness"[13], describes a test that can be used to determine whether an application can reliably establish sessions quickly in a complex environment such as dual-stack (IPv4+IPv6) deployment or IPv6 deployment with multiple prefixes and upstream ingress filtering.

DNS LANGUAGE LIBRARY CONSIDERATIONS

Depending on the language used, there may be some changes necessary to conduct DNS lookups. The changes range from significant reworking of code to simple configuration changes.

The traditional system call used by C and C++ to conduct DNS lookups is gethostbyname(). Over the years, several other functions such as gethostbyname2(), getipnodebyname(), and getipnodebyaddr() have also been added. However, all of these functions have their issues and are no longer considered usable for IPv6 networking. The getaddrinfo() system call replaces them all, and is much better suited for DNS lookups in IPv4 and IPv6 environments.

On many platforms, getaddrinfo() can take the AI_ADDRCONFIG flag, which instructs the function to only obtain addresses that are usable. For example, if IPv6 wasn't enabled in your system, getaddrinfo() wouldn't query for AAAA records, since your host wouldn't be able to use the IPv6 addresses. Unfortunately, AI_ADDRCONFIG isn't supported on all platforms.[14]

Java has supported IPv6 since version 1.4. By default, Sun's JVM will run in dual-stack mode. However, even in dual-stack mode, the JVM prefers to use A records when performing name resolution, but you can set it to prefer AAAA records by setting:

-Djava.net.preferIPv6Addresses=true

In Python the socket module provides access to the BSD socket interface. When representing a socket address, if you use a hostname in the host portion of IPv4/IPv6 socket address, the program may show a non-deterministic behavior, as Python uses the first address returned from the DNS resolution. The socket address will be resolved differently into an actual IPv4/IPv6 address, depending on the results from DNS resolution and/or the host configuration. For deterministic behavior use a numeric address in host portion.[15]

This module includes a function socket.getaddrinfo(host, port[, family[, socktype[, proto[, flags]]]]) which translates the host/port argument into a sequence of quinttuples that contain all the necessary arguments for creating a socket connected to that service. This function supports both IPv4 and IPv6 hosts.

The socket Python module also includes a function socket.gethostbyname(hostname) which translates a hostname into an IPv4 address. It's worth noting that this function doesn't support IPv6 name resolution and getaddrinfo() should be used instead, just as with C and C++. The same is true of the socket.gethostbyname_ex(hostname) function; use getaddrinfo() instead.

Lastly, the Python function socket.gethostbyaddr(ip_address) returns the primary hostname responding to the given IP address, a list of alternative hostnames for the same address and a list of IP addresses for the same interface on the same host. This function supports both IPv4 and IPv6.

12 http://tools.ietf.org/html/rfc6555
13 http://tools.ietf.org/html/rfc6556
14 https://wikispaces.psu.edu/display/ipv6/IPv6+programming
15 https://docs.python.org/2/library/socket.html

MISCELLANEOUS "GOTCHAS"

As good as state-of-the-art development tools are, modern application development can be complex consisting of many systems and components. And preparing all these components for the IPv6 Internet requires looking at how they communicate, how they share information, and what data they store and more.

Though far from complete, here are a couple of areas that might bear investigation:

1) Content Delivery Networks (CDNs) - Some applications rely on CDNs for delivery of large files in a more performant manner than self-hosting. If your application uses CDNs, you may want to consider the following:
    a) Does the CDN serve content over IPv6? If it doesn't how will the content be proxied between its IPv4 servers and your IPv6 customer?
    b) Have AAAA DNS records been provisioned for serving CDN data to IPv6 clients?
2) Application Programming Interfaces (APIs) - APIs take many shapes and forms, including SOAP messages, URIs, native library calls, etc. If your application publishes an API or uses an API in which an API call takes an IP address as a parameter, you should check that the parameter can be passed as either an IPv4 address or an IPv6 address.
3) Virtual Machines and System Containers and Clouds - Virtual machines and lighter-weight system containers are a commonly used mechanism to quickly achieve scalability, especially in cloud computing. However not every virtual machine is capable of IPv6 connectivity. If your system relies on these, you should check to see that your host and any virtual machines hosted within support IPv6.
4) The Obvious But Overlooked - By now the implications of IP addresses appearing in multiple areas should be apparent, but it does not hurt to mention some that are routinely overlooked:
    a) Configuration files and other configuration data
    b) Monitoring and alert systems
    c) Log analyzers and analytic engines
5) Documentation - Finally, documentation for software will need modification to include IPv6 addresses. The IETF has set aside 2001:db8::/32 as a set of IPv6 addresses to be used as examples in documentation. These addresses should never be routed on the Internet, and therefore their use in examples is harmless.

CONCLUSION

The impending depletion of IPv4 addresses does not necessarily mean the Internet will see an immediate shift to IPv6. Over time the use of IPv4 addresses will become more costly for network operators with respect to ownership and network infrastructure. The move to IPv6 will not happen overnight, but pressure will build until there is a tipping point.

Fortunately, software developers still have time to make the needed changes for the transition. The goal of this document is to help make the development lifecycle for those changes easier to plan and less subject to trial-and-error.

We, the authors, do not claim to have a comprehensive list of software changes needed for IPv6 transition, but we do feel this should give you a very good start in any software migration endeavors. Should you have comments, questions, or have discovered other topics that should warrant attention, feel free to contact us.

AUTHORS

Andy Newton is the Chief Engineer for the American Registry for Internet Numbers (ARIN), responsible for software/systems engineering, standards development and research. Andy has many years of software engineering experience ranging from embedded systems, thick clients, and application servers in a broad array of areas such as industrial controls, back office and work flow processes, Voice-over-IP and more. Andy also holds a black-belt rank in Judo. Andy can be reached at andy@arin.net.

Sofía Silva Berenguer is the former Senior Security and Stability Specialist for the Internet Address Registry for Latin America and the Caribbean (LACNIC), where she performed research into the Domain Name System Security Extensions (DNSSEC), the Resource Public Key Infrastructure (RPKI), anycast services, and Internet Exchange Points (IXPs). She is currently conducting in-depth network research as a Research Assistant for IMDEA Networks in Madrid, Spain, and a graduate student in Masters in Telematics Engineering at the University Carlos III of Madrid.