Twitter Thread by Dave Anderson



Dave Anderson
@dave_universetf

In my quest to write a fast IPv4+6 parser, I have written a slow-but-I-think-correct parser, to use as a base of comparison.

In doing so, I have discovered more cursed IP address representations that I was previously unaware of.

A thread!

We start out simple, with IPv4 and IPv6 in what I'll call their "canonical form":

192.168.0.1
1:2:3:4:5:6:7:8

Various specs call these "dotted quad", dot-separated fields each representing 1 byte; and "colon-hex", colon-separated fields each representing 2 bytes.

The first bits of complexity come from IPv6. In canonical form, common addresses would end up with long runs of zeros in the middle. So, "::" allows you to elide 1 or more 16-bit blocks of zeros:

1:2::3:4 means 1:2:0:0:0:0:3:4

Next up, for cursed historical reasons, IPv6 permits you to write the final 32 bits of the address in dotted quad form. Effectively, you can splat an IPv4 address onto the end of IPv6 addresses!

1:2:3:4:5:6:77.77.88.88 means 1:2:3:4:5:6:7777:8888

And of course, you can combine the two!

fe80::1.2.3.4 means fe80:0:0:0:0:0:102:304

The existence of :: also introduces an annoying edge case in parsing: the "::" can be at the start or end of the address, and the "empty" side of the address is not one of the 16-bit fields.

::1 means 0:0:0:0:0:0:0:1
1:: means 1:0:0:0:0:0:0:0
:: means 0:0:0:0:0:0:0:0

That's a natural consequence of the :: rule, but it makes the parsers slightly more annoying to write.

One final rule for IPv6: technically, each colon-hex field is 4 hex digits, but you can elide leading zeros.

Fully canonically, :: is 0000:0000:0000:000:0000:0000:0000:0000

But we allow the compacted forms.

My apologies to my trypophobic followers.

That's it for IPv6, mostly. Now, on to IPv4!

Fun fact, the textual representation of IPv4 was never standardized in any document before IPv6 needed a grammar for its weirdo "trailing dotted quad" notation.

So, it's a de-facto standard that boils down to mostly "what did 4.2BSD understand?"

And hoo boy, strap yourselves in, because 4.2BSD sure had some whacky opinions!

Let's use 192.168.140.255 as an example. That's an IPv4 that people would look at and go "yes, that sure is an IPv4 address."

How else can we write that exact same address?

This is the same IP address: 3232271615

You get that by simply interpreting the 4 bytes of the IP address as a big-endian unsigned 32-bit integer, and print that.

If you visit http://3232271615 , Chrome will attempt to load http://192.168.140.255.

Okay, but that's sort-of sensible, right? An IPv4 address is 4 bytes, so printing it as a single number is a bit human-unfriendly, but broadly plausible, right?

Okay, how about 0300.0250.0214.0377 ?

Yup, that's the same address. Dotted quad, except each field is written out in octal.

If octal is supported, you might wonder about hex.

And you'd be right! 192.168.140.255 is also 0xc0.0xa8.0x8c.0xff , according to 4.2BSD.

Now, remember before we had CIDR (Classless Inter-Domain Routing) ? IPv4 addresses were Class A, Class B or Class C. It was a weird time.

And that weird time made it into IP addresses!

192.168.140.255 is the "Class C" notation.

192.168.36095 is the "Class B" notation.
192.11046143 is the "Class A" notation.

Basically, coalesce the final fields into either a 16-bit or a 24-bit integer field, because why not.

And finally, we come to one last bit of unspecified behavior: do IPv4 addresses permit an unlimited number of leading zeros in each quad? Or is there a maximum of 3 digits?

001.002.003.004 is universally recognized as valid. What about 0000000001.0000000002.0000000003.000000004?

You might be wondering if either of these numbers should be read in as octal, per one of the previous bits of this thread.

It depends! There are implementations that do either, but _most_ modern implementations have abandoned the octal notation and treat leading 0s as decimal.

Oh, and the leading zero debate also infects IPv6, to some extent! The specs tried to specify the textual representation of IPv6, but it failed to be complete. So it's unclear if 000001::00001.00002.00003.00004 is a valid IPv6 address ("common" form 1::1.2.3.4, or 1::102:304).

Most modern parsers seem to allow an unlimited amount of leading zeros in their representations, probably because they're leaning on some "parse integer" library that implements that behavior.

And so, we reach the bitter end. If you want to _truly_ parse IP addresses, this is the bullshit you have to put up with.

Currently, my slow reference parser jettisons a lot of old baggage, and sticks to what I think is a sensible subset of these possibilities.

My parser understands classic v4 dotted quad, with any number of leading zeros. It does not process Class A/B notation, or hex or octal notation. It does not process the "uint32 to the knee" representation.

For IPv6, it understands canonical colon-hex form, as well as :: and trailing-IPv4 style (where the trailing IPv4 follows the same rules as the previous tweet). Each field is allowed any number of leading zeros.

And as @alanjmcf noticed, I messed up one of the representations above.

1:2:3:4:5:6:77.77.88.88 means 1:2:3:4:5:6:4d4d:5858, not 1:2:3:4:5:6:7777:8888. I missed out a decimal-to-hex conversion in there.

This thread is now https://t.co/PlJgkNlOqz , with a little rewording, but same content, if you want to handily link it beyond twitter.