3. IP Addresses, `structs`, and Data Munging

Here's the part of the game where we get to talk code for a change.

But first, let's discuss more non-code! Yay! First I want to talk about IP addresses and ports for just a tad so we have that sorted out. Then we'll talk about how the sockets API stores and manipulates IP addresses and other data.

3.1. IP Addresses, versions 4 and 6

In the good old days back when Ben Kenobi was still called Obi Wan Kenobi, there was a wonderful network routing system called The Internet Protocol Version 4, also called IPv4. It had addresses made up of four bytes (A.K.A. four "octets"), and was commonly written in "dots and numbers" form, like so: `192.0.2.111`.

You've probably seen it around.

In fact, as of this writing, virtually every site on the Internet uses IPv4.

Everyone, including Obi Wan, was happy. Things were great, until some naysayer by the name of Vint Cerf warned everyone that we were about to run out of IPv4 addresses!

(Besides warning everyone of the Coming IPv4 Apocalypse Of Doom And Gloom, Vint Cerf is also well-known for being The Father Of The Internet. So I really am in no position to second-guess his judgment.)

Run out of addresses? How could this be? I mean, there are like billions of IP addresses in a 32-bit IPv4 address. Do we really have billions of computers out there?

Yes.

Also, in the beginning, when there were only a few computers and everyone thought a billion was an impossibly large number, some big organizations were generously allocated millions of IP addresses for their own use. (Such as Xerox, MIT, Ford, HP, IBM, GE, AT&T, and some little company called Apple, to name a few.)

In fact, if it weren't for several stopgap measures, we would have run out a long time ago.

But now we're living in an era where we're talking about every human having an IP address, every computer, every calculator, every phone, every parking meter, and (why not) every puppy dog, as well.

And so, IPv6 was born. Since Vint Cerf is probably immortal (even if his physical form should pass on, heaven forbid, he is probably already existing as some kind of hyper-intelligent ELIZA program out in the depths of the Internet2), no one wants to have to hear him say again "I told you so" if we don't have enough addresses in the next version of the Internet Protocol.

What does this suggest to you?

~~Google Adsense~~

That we need a *lot* more addresses. That we need not just twice as many addresses, not a billion times as many, not a thousand trillion times as many, but *79 MILLION BILLION TRILLION times as many possible addresses!* That'll show 'em!

You're saying, "Beej, is that true? I have every reason to disbelieve large numbers." Well, the difference between 32 bits and 128 bits might not sound like a lot; it's only 96 more bits, right? But remember, we're talking powers here: 32 bits represents some 4 billion numbers (2^{32}), while 128 bits represents about 340 trillion trillion trillion numbers (for real, 2^{128}). That's like a million IPv4 Internets for *every single star in the Universe*.

Forget this dots-and-numbers look of IPv4, too; now we've got a hexadecimal representation, with each two-byte chunk separated by a colon, like this: `2001:0db8:c9d2:aee5:73e3:934a:a5ae:9551`.

That's not all! Lots of times, you'll have an IP address with lots of zeros in it, and you can compress them between two colons. And you can leave off leading zeros for each byte pair. For instance, each of these pairs of addresses are equivalent:

```
2001:0db8:c9d2:0012:0000:0000:0000:0051
2001:db8:c9d2:12::51

2001:0db8:ab00:0000:0000:0000:0000:0000
2001:db8:ab00::

0000:0000:0000:0000:0000:0000:0000:0001
::1
```

The address `::1` is the *loopback address*. It always means "this machine I'm running on now". In IPv4, the loopback address is 127.0.0.1.

Finally, there's an IPv4-compatibility mode for IPv6 addresses that you might come across. If you want, for example, to represent the IPv4 address 192.0.2.33 as an IPv6 address, you use the following notation: "`::ffff:192.0.2.33`".

We're talking serious fun.

In fact, it's such serious fun, that the Creators of IPv6 have quite cavalierly lopped off trillions and trillions of addresses for reserved use, but we have so many, frankly, who's even counting anymore? There are plenty left over for every man, woman, child, puppy, and parking meter on every planet in the galaxy. And believe me, every planet in the galaxy has parking meters. You know it's true.

3.1.1. Subnets

For organizational reasons, it's sometimes convenient to declare that "this first part of this IP address up through this bit is the *network portion* of the IP address, and the remainder is the *host portion*.

For instance, with IPv4, you might have `192.0.2.12`, and we could say that the first three bytes are the network and the last byte was the host. Or, put another way, we're talking about host `12` on network `192.0.2.0` (see how we zero out the byte that was the host.)

And now for more outdated information! Ready? In the Ancient Times, there were "classes" of subnets, where the first one, two, or three bytes of the address was the network part. If you were ~~Google Adsense~~ lucky enough to have one byte for the network and three for the host, you could have 24 bits-worth of hosts on your network (24 million or so). That was a "Class A" network. On the opposite end was a "Class C", with three bytes of network, and one byte of host (256 hosts, minus a couple that were reserved.)

So as you can see, there were just a few Class As, a huge pile of Class Cs, and some Class Bs in the middle.

The network portion of the IP address is described by something called the *netmask*, which you bitwise-AND with the IP address to get the network number out of it. The netmask usually looks something like `255.255.255.0`. (E.g. with that netmask, if your IP is `192.0.2.12`, then your network is `192.0.2.12` AND `255.255.255.0` which gives `192.0.2.0`.)

Unfortunately, it turned out that this wasn't fine-grained enough for the eventual needs of the Internet; we were running out of Class C networks quite quickly, and we were most definitely out of Class As, so don't even bother to ask. To remedy this, The Powers That Be allowed for the netmask to be an arbitrary number of bits, not just 8, 16, or 24. So you might have a netmask of, say `255.255.255.252`, which is 30 bits of network, and 2 bits of host allowing for four hosts on the network. (Note that the netmask is *ALWAYS* a bunch of 1-bits followed by a bunch of 0-bits.)

But it's a bit unwieldy to use a big string of numbers like `255.192.0.0` as a netmask. First of all, people don't have an intuitive idea of how many bits that is, and secondly, it's really not compact. So the New Style came along, and it's much nicer. You just put a slash after the IP address, and then follow that by the number of network bits in decimal. Like this: `192.0.2.12/30`.

Or, for IPv6, something like this: `2001:db8::/32` or `2001:db8:5413:4028::9db9/64`.

3.1.2. Port Numbers

If you'll kindly remember, I presented you earlier with the Layered Network Model which had the Internet Layer (IP) split off from the Host-to-Host Transport Layer (TCP and UDP). Get up to speed on that before the next paragraph.

Turns out that besides an IP address (used by the IP layer), there is another address that is used by TCP (stream sockets) and, coincidentally, by UDP (datagram sockets). It is the *port number*. It's a 16-bit number that's like the local address for the connection.

Think of the IP address as the street address of a hotel, and the port number as the room number. That's a decent analogy; maybe later I'll come up with one involving the automobile industry.

Say you want to have a computer that handles incoming mail AND web services—how do you differentiate between the two on a computer with a single IP address?

Well, different services on the Internet have different well-known port numbers. You can see them all in the Big IANA Port List or, if you're on a Unix box, in your `/etc/services` file. HTTP (the web) is port 80, telnet is port 23, SMTP is port 25, the game DOOM used port 666, etc. and so on. Ports under 1024 are often considered special, and usually require special OS privileges to use.

And that's about it!

3.2. Byte Order

By Order of the Realm! There shall be two byte orderings, hereafter to be known as Lame and Magnificent!

I joke, but one really is better than the other. `:-)`

There really is no easy way to say this, so I'll just blurt it out: your computer might have been storing bytes in reverse order behind your back. I know! No one wanted to have to tell you.

The thing is, everyone in the Internet world has generally agreed that if you want to represent the two-byte hex number, say `b34f`, you'll store it in two sequential bytes `b3` followed by `4f`. Makes sense, and, as Wilford Brimley would tell you, it's the Right Thing To Do. This number, stored with the big end first, is called *Big-Endian*.

Unfortunately, a few computers scattered here and there throughout the world, namely anything with an Intel or Intel-compatible processor, store the bytes reversed, so `b34f` would be stored in memory as the sequential bytes `4f` followed by `b3`. This storage method is called *Little-Endian*.

But wait, I'm not done with terminology yet! The more-sane *Big-Endian* is also called *Network Byte Order* because that's the order us network types like.

Your computer stores numbers in *Host Byte Order*. If it's an Intel 80x86, Host Byte Order is Little-Endian. If it's a Motorola 68k, Host Byte Order is Big-Endian. If it's a PowerPC, Host Byte Order is... well, it depends!

A lot of times when you're building packets or filling out data structures you'll need to make sure your two- and four-byte numbers are in Network Byte Order. But how can you do this if you don't know the native Host Byte Order?

Good news! You just get to assume the Host Byte Order isn't right, and you always run the value through a function to set it to Network Byte Order. The function will do the magic conversion if it has to, and this way your code is portable to machines of differing endianness.

All righty. There are two types of numbers that you can convert: `short` (two bytes) and `long` (four bytes). These functions work for the `unsigned` variations as well. Say you want to convert a `short` from Host Byte Order to Network Byte Order. Start with "h" for "host", follow it with "to", then "n" for "network", and "s" for "short": h-to-n-s, or **`htons()`** (read: "Host to Network Short").

It's almost too easy...

You can use every combination of "n", "h", "s", and "l" you want, not counting the really stupid ones. For example, there is NOT a **`stolh()`** ("Short to Long Host") function—not at this party, anyway. But there are:

htons()	**h**ost **to** **n**etwork **s**hort
htonl()	**h**ost **to** **n**etwork **l**ong
ntohs()	**n**etwork **to** **h**ost **s**hort
ntohl()	**n**etwork **to** **h**ost **l**ong

~~Google Adsense~~

Basically, you'll want to convert the numbers to Network Byte Order before they go out on the wire, and convert them to Host Byte Order as they come in off the wire.

I don't know of a 64-bit variant, sorry. And if you want to do floating point, check out the section on Serialization, far below.

Assume the numbers in this document are in Host Byte Order unless I say otherwise.

3.3. structs

Well, we're finally here. It's time to talk about programming. In this section, I'll cover various data types used by the sockets interface, since some of them are a real bear to figure out.

First the easy one: a socket descriptor. A socket descriptor is the following type:

```
int
```

Just a regular `int`.

Things get weird from here, so just read through and bear with me.

My First Struct™—`struct addrinfo`. This structure is a more recent invention, and is used to prep the socket address structures for subsequent use. It's also used in host name lookups, and service name lookups. That'll make more sense later when we get to actual usage, but just know for now that it's one of the first things you'll call when making a connection.

```
struct addrinfo {
    int              ai_flags;     // AI_PASSIVE, AI_CANONNAME, etc.
    int              ai_family;    // AF_INET, AF_INET6, AF_UNSPEC
    int              ai_socktype;  // SOCK_STREAM, SOCK_DGRAM
    int              ai_protocol;  // use 0 for "any"
    size_t           ai_addrlen;   // size of ai_addr in bytes
    struct sockaddr *ai_addr;      // struct sockaddr_in or _in6
    char            *ai_canonname; // full canonical hostname

    struct addrinfo *ai_next;      // linked list, next node
};
```

You'll load this struct up a bit, and then call **getaddrinfo()**. It'll return a pointer to a new linked list of these structures filled out with all the goodies you need.

You can force it to use IPv4 or IPv6 in the *ai_family* field, or leave it as `AF_UNSPEC` to use whatever. This is cool because your code can be IP version-agnostic.

Note that this is a linked list: *`ai_next`* points at the next element—there could be several results for you to choose from. I'd use the first result that worked, but you might have diffe ~~Google Adsense~~ needs; I don't know everything, man!

You'll see that the *`ai_addr`* field in the `struct addrinfo` is a pointer to a `struct sockaddr`. This is where we start getting into the nitty-gritty details of what's inside an IP address structure.

You might not usually need to write to these structures; oftentimes, a call to **`getaddrinfo()`** to fill out your `struct addrinfo` for you is all you'll need. You *will*, however, have to peer inside these `struct`s to get the values out, so I'm presenting them here.

(Also, all the code written before `struct addrinfo` was invented packed all this stuff by hand, so you'll see a lot of IPv4 code out in the wild that does exactly that. You know, in old versions of this guide and so on.)

Some `struct`s are IPv4, some are IPv6, and some are both. I'll make notes of which are what.

Anyway, the `struct sockaddr` holds socket address information for many types of sockets.

```
struct sockaddr {
    unsigned short    sa_family;    // address family, AF_xxx
    char              sa_data[14];  // 14 bytes of protocol address
};
```

`sa_family` can be a variety of things, but it'll be `AF_INET` (IPv4) or `AF_INET6` (IPv6) for everything we do in this document. *`sa_data`* contains a destination address and port number for the socket. This is rather unwieldy since you don't want to tediously pack the address in the *`sa_data`* by hand.

To deal with `struct sockaddr`, programmers created a parallel structure: `struct sockaddr_in` ("in" for "Internet") to be used with IPv4.

And *this is the important* bit: a pointer to a `struct sockaddr_in` can be cast to a pointer to a `struct sockaddr` and vice-versa. So even though **`connect()`** wants a `struct sockaddr*`, you can still use a `struct sockaddr_in` and cast it at the last minute!

```
// (IPv4 only--see struct sockaddr_in6 for IPv6)

struct sockaddr_in {
    short int          sin_family;  // Address family, AF_INET
    unsigned short int sin_port;    // Port number
    struct in_addr     sin_addr;    // Internet address
    unsigned char      sin_zero[8]; // Same size as struct sockaddr
};
```

This structure makes it easy to reference elements of the socket address. Note that *`sin_zero`* (which is included to pad the structure to the length of a `struct sockaddr`) should be set to all zeros with the function **`memset()`**. Also, notice that *`sin_family`* corresponds to *`sa_family`* in a `struct sockaddr` and should be set to "`AF_INET`". Finally, the *`sin_port`* must be in *Network Byte Order* (by using **`htons()`**!)

Let's dig deeper! You see the *`sin_addr`* field is a `struct in_addr`. What is that thing? Well, not to be overly dramatic, but it's one of the scariest unions of all time:

~~Google Adsense~~

```
// (IPv4 only--see struct in6_addr for IPv6)

// Internet address (a structure for historical reasons)
struct in_addr {
    uint32_t s_addr; // that's a 32-bit int (4 bytes)
};
```

Whoa! Well, it *used* to be a union, but now those days seem to be gone. Good riddance. So if you have declared *ina* to be of type `struct sockaddr_in`, then *ina.sin_addr.s_addr* references the 4-byte IP address (in Network Byte Order). Note that even if your system still uses the God-awful union for `struct in_addr`, you can still reference the 4-byte IP address in exactly the same way as I did above (this due to `#define`s.)

What about IPv6? Similar `struct`s exist for it, as well:

```
// (IPv6 only--see struct sockaddr_in and struct in_addr for IPv4)

struct sockaddr_in6 {
    u_int16_t       sin6_family;   // address family, AF_INET6
    u_int16_t       sin6_port;     // port number, Network Byte Order
    u_int32_t       sin6_flowinfo; // IPv6 flow information
    struct in6_addr sin6_addr;     // IPv6 address
    u_int32_t       sin6_scope_id; // Scope ID
};

struct in6_addr {
    unsigned char   s6_addr[16];   // IPv6 address
};
```

Note that IPv6 has an IPv6 address and a port number, just like IPv4 has an IPv4 address and a port number.

Also note that I'm not going to talk about the IPv6 flow information or Scope ID fields for the moment... this is just a starter guide. :-)

Last but not least, here is another simple structure, `struct sockaddr_storage` that is designed to be large enough to hold both IPv4 and IPv6 structures. (See, for some calls, sometimes you don't know in advance if it's going to fill out your `struct sockaddr` with an IPv4 or IPv6 address. So you pass in this parallel structure, very similar to `struct sockaddr` except larger, and then cast it to the type you need:

```
struct sockaddr_storage {
    sa_family_t  ss_family;     // address family

    // all this is padding, implementation specific, ignore it:
    char      __ss_pad1[_SS_PAD1SIZE];
    int64_t   __ss_align;
    char      __ss_pad2[_SS_PAD2SIZE];
};
```

What's important is that you can see the address family in the *ss_family* field—check this to see if it's `AF_INET` or `AF_INET6` (for IPv4 or IPv6). Then you can cast it to a `struct sockaddr_in` or `struct sockaddr_in6` if you wanna.

3.4. IP Addresses, Part Deux

~~Google Adsense~~

Fortunately for you, there are a bunch of functions that allow you to manipulate IP addresses. No need to figure them out by hand and stuff them in a `long` with the `<<` operator.

First, let's say you have a `struct sockaddr_in ina`, and you have an IP address "`10.12.110.57`" or "`2001:db8:63b3:1::3490`" that you want to store into it. The function you want to use, **`inet_pton()`**, converts an IP address in numbers-and-dots notation into either a `struct in_addr` or a `struct in6_addr` depending on whether you specify `AF_INET` or `AF_INET6`. ("pton" stands for "presentation to network"—you can call it "printable to network" if that's easier to remember.) The conversion can be made as follows:

```
struct sockaddr_in sa; // IPv4
struct sockaddr_in6 sa6; // IPv6

inet_pton(AF_INET, "192.0.2.1", &(sa.sin_addr)); // IPv4
inet_pton(AF_INET6, "2001:db8:63b3:1::3490", &(sa6.sin6_addr)); // IPv6
```

(Quick note: the old way of doing things used a function called **`inet_addr()`** or another function called **`inet_aton()`**; these are now obsolete and don't work with IPv6.)

Now, the above code snippet isn't very robust because there is no error checking. See, **`inet_pton()`** returns `-1` on error, or 0 if the address is messed up. So check to make sure the result is greater than 0 before using!

All right, now you can convert string IP addresses to their binary representations. What about the other way around? What if you have a `struct in_addr` and you want to print it in numbers-and-dots notation? (Or a `struct in6_addr` that you want in, uh, "hex-and-colons" notation.) In this case, you'll want to use the function **`inet_ntop()`** ("ntop" means "network to presentation"—you can call it "network to printable" if that's easier to remember), like this:

```
// IPv4:

char ip4[INET_ADDRSTRLEN];  // space to hold the IPv4 string
struct sockaddr_in sa;      // pretend this is loaded with something

inet_ntop(AF_INET, &(sa.sin_addr), ip4, INET_ADDRSTRLEN);

printf("The IPv4 address is: %s\n", ip4);


// IPv6:

char ip6[INET6_ADDRSTRLEN]; // space to hold the IPv6 string
struct sockaddr_in6 sa6;    // pretend this is loaded with something

inet_ntop(AF_INET6, &(sa6.sin6_addr), ip6, INET6_ADDRSTRLEN);

printf("The address is: %s\n", ip6);
```

When you call it, you'll pass the address type (IPv4 or IPv6), the address, a pointer to a string to hold the result, and the maximum length of that string. (Two macros conveniently hold the size of the string you'll need to hold the largest IPv4 or IPv6 address: `INET_ADDRSTRLEN` and `INET6_ADDRSTRLEN`.)

(Another quick note to mention once again the old way of doing things: the historical function to do this conversion was called **`inet_ntoa()`**. It's also obsolete and won't work with IPv6.) ~~Google Adsense~~

Lastly, these functions only work with numeric IP addresses—they won't do any nameserver DNS lookup on a hostname, like "www.example.com". You will use **`getaddrinfo()`** to do that, as you'll see later on.

3.4.1. Private (Or Disconnected) Networks

Lots of places have a firewall that hides the network from the rest of the world for their own protection. And often times, the firewall translates "internal" IP addresses to "external" (that everyone else in the world knows) IP addresses using a process called *Network Address Translation*, or NAT.

Are you getting nervous yet? "Where's he going with all this weird stuff?"

Well, relax and buy yourself a non-alcoholic (or alcoholic) drink, because as a beginner, you don't even have to worry about NAT, since it's done for you transparently. But I wanted to talk about the network behind the firewall in case you started getting confused by the network numbers you were seeing.

For instance, I have a firewall at home. I have two static IPv4 addresses allocated to me by the DSL company, and yet I have seven computers on the network. How is this possible? Two computers can't share the same IP address, or else the data wouldn't know which one to go to!

The answer is: they don't share the same IP addresses. They are on a private network with 24 million IP addresses allocated to it. They are all just for me. Well, all for me as far as anyone else is concerned. Here's what's happening:

If I log into a remote computer, it tells me I'm logged in from 192.0.2.33 which is the public IP address my ISP has provided to me. But if I ask my local computer what it's IP address is, it says 10.0.0.5. Who is translating the IP address from one to the other? That's right, the firewall! It's doing NAT!

10.*x*.*x*.*x* is one of a few reserved networks that are only to be used either on fully disconnected networks, or on networks that are behind firewalls. The details of which private network numbers are available for you to use are outlined in RFC 1918, but some common ones you'll see are 10.*x*.*x*.*x* and 192.168.*x*.*x*, where *x* is 0-255, generally. Less common is 172.*y*.*x*.*x*, where *y* goes between 16 and 31.

Networks behind a NATing firewall don't *need* to be on one of these reserved networks, but they commonly are.

(Fun fact! My external IP address isn't really 192.0.2.33. The 192.0.2.*x* network is reserved for make-believe "real" IP addresses to be used in documentation, just like this guide! Wowzers!)

IPv6 has private networks, too, in a sense. They'll start with `fd`*`xx`*`:` (or maybe in the future `fc`*`XX`*`:`), as per RFC 4193. NAT and IPv6 don't generally mix, however (unless you're doing the IPv6 to IPv4 gateway thing which is beyond the scope of this document)—in theory you'll have so many addresses at your disposal that you won't need to use NAT any longer. But if you want to allocate addresses for yourself on a network that won't route outside, this is how to do it.

<< Prev Beej's Guide to Network Programming Next >>

~~Google Adsense~~