A Guide to IP Subnetting

1) The Basics of Binary

Counting In Binary

In order to understand IP subnetting it is important to first understand how to count in binary. As a first step, lets start with something familiar; counting in decimal.

Consider the progression of the following number sequence:

```
00
01
02
..
08
09
10
11
12
..
18
19
20
```

In decimal, there are 10 characters available for use: the numbers 0 through 9. Starting at zero, we count by incrementing the first column by one until we reach the final character of 9. At this point the first column resets to zero and the second column is incremented by one. We then iterate the characters in the first column until we again reach 9, at which point we repeat the process.

Counting in binary works identically. The only difference is that in binary we only have two characters available: the numbers 0 and 1. Consider the following binary number sequence. See if you can spot the pattern. The decimal conversion is represented within the () next to each binary sequence.

```
0000 (0)
0001 (1)
0010 (2)
0011 (3)
0100 (4)
..
1100 (12)
1101 (13)
1110 (14)
```

```
1111 (15)
```

Quickly Converting Between Binary and Decimal

Quickly, what is binary 1100 1010 in decimal? If you are unfamiliar with the shortcut, then you probably took the approach of starting a 0000 0000 and counting up. This approach is painful. Luckily there is an easier way if you keep the following fact in mind: each column in a binary sequence has a place value. This is similar to decimal, where each column has a place value which is a mulltiple of 10. In binary, the difference is that columns have place values which are powers of 2. The following table illustrates this:

128	**64**	**32**	**16**	**8**	**4**	**2**	**1**
1	1	0	0	1	0	1	0

As seen above, each column is assigned the value which is a power of 2. In other words, each column is double the value of the previous column. I have mapped the number 1100 1010 into this table. Using the place values of each column we can convert to decimal simply by adding the value of each column where a “1” is present. In this case we have 128 + 64 + 8 + 2 = 202. Easy.

In order to convert decimal to binary we would use a similar approach. If I want to convert 202 to binary then I would start subtracting large powers of 2 from 202 and tracking them in a table. So:

1. Take 202 - 128 = 74. Mark a “1” in the 128 column.
2. Take 74 - 64 = 10. Mark a “1” in the 64 column.
3. Since 10 is less than 32 and 16 we can’t use those, but we can do 10 - 8 = 2. Mark a “1” in the 8 column.
4. Since 2 is less than 4 we can’t use a 4, but we can do 2 - 2 = 0. Mark a “1” in the 2 column.

Done. We end up with the same table as above:

128	**64**	**32**	**16**	**8**	**4**	**2**	**1**
1	1	0	0	1	0	1	0

2) IP Addressing

An IP addres is simply a large integer, where IPv4 has 32 bits of capacity and IPv6 has 128 bits of capacity. Lets see an example of an IPv4 address as a binary number.

```
0111 1111 0000 0000 0000 0000 0000 0001
```

Imagine how difficult it would be to configure an IP address if we always represented them in binary. Instead, IP addresses are typically written using something other than binary. For IPv4, the most commonly used notation is that of a 4 groups of 8 bits, converted to decimal, and separated by periods. Using the example above:

```
0111 1111 0000 0000 0000 0000 0000 0001
0111 1111 . 0000 0000 . 0000 0000 . 0000 0001
127.0.0.1
```

Lets see an IPv6 example:

```
1111 1111 1111 1111 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0001
```

Thats ugly. Hopefully I didnt miss any bits in that huge string of digits. Luckily IPv6 uses hex as shorthand so that the addressing is a bit more managable. Specifically, IPv6 uses groups of 16 bits broken up by colons. Using the above example:

```
1111 1111 1111 1111 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0001
1111 1111 1111 1111 : 0000 0000 0000 0000 : 0000 0000 0000 0000 : 0000 0000 0000
0000 : 0000 0000 0000 0000 : 0000 0000 0000 0000 : 0000 0000 0000 0000 : 0000
0000 0000 0001
ffff:0:0:0:0:0:1
```

IPv6 takes an additional step of compressing large sequences of zeros into a “::” (note that you can only use a single “::” in a given address). So the address above could be written as ffff::1.

3) IP Subnetting

In order to more effectively utilize the large 32 or 128 bit address space of IPv4 and IPv6, the designers of these protocols introduced the idea of subnetting. Subnetting may be though of as a means of breaking the entire collective of IP addresses into groups of smaller contiguous addresses (or networks). Subnetting allows us to break up the IP addressing space into smaller groups which may then be allocated to individual organizations. In turn, these organizations may utilize subnetting to further break up their allocated address space into yet smaller blocks for use within their local networks.

In order to represent a subnet there are 2 pieces of data required: an IP address and a subnet mask. The purpose of the mask is to deliniate the “network” portion of the address from the “host” portion of the address. Lets consider one of the most commonly size networks in use today; the /24. If I were to allocate the network 10.0.0.0/24 to you, then you would probably understand that you have been given the range of addresses 10.0.0.0 - 10.0.0.255. What you may not understand is the mechanism for determining this range of addresses.

Firstly, lets consider the subnet mask “/24” from the network 10.0.0.0/24. The subnet mask tells us that 24 bits are reserved for the “network” portion of the address while the remaining 8 bits (32 - 24 = 8) are reserved for the “host” portion. Knowing the range of IP addresses owned by this block is a matter of understanding that the “network” portion of the address is fixed (i.e. identical for all

addresses on the network) and that only the “host” portion is variable per host on the network. Lets explore this further by representing both the subnet mask and the network address in binary:

```
mask: 1111 1111 . 1111 1111 . 1111 1111 . 0000 0000
address: 0000 1010 . 0000 0000 . 0000 0000 . 0000 0000
```

In the above breakout we see that the first 24 bits of the total 32 bits are represented by 1’s. This matches the definition of a /24 subnet mask and tells us that all addresses in this network must share the same 24 bits. Lets iterate through some addresses of this network. In the below table I separate the network and host portions of each address. The #1 rule is that we cannot modify any bits from the network portion.

network bits	**host bits**	**address**
0000 1010 . 0000 0000 . 0000 0000	0000 0000	10.0.0.0
0000 1010 . 0000 0000 . 0000 0000	0000 0001	10.0.0.1
0000 1010 . 0000 0000 . 0000 0000	0000 0010	10.0.0.2
…	…	…
0000 1010 . 0000 0000 . 0000 0000	1111 1110	10.0.0.254
0000 1010 . 0000 0000 . 0000 0000	1111 1111	10.0.0.255

As seen above, once all bits within the host portion have been set to “1” then we have reached the final address of the network. In the example above we see that the range of addresses for this network are 10.0.0.0 - 10.0.0.255, or 256 addresses (2 to the 8th power).

Lets look at an example using something other than a /24. Assume that we wanted to subnet 10.0.0.0/24 into a /30 instead. Here is the breakdown:

network bits	**host bits**	**address**
0000 1010 . 0000 0000 . 0000 0000 . 0000 00	00	10.0.0.0
0000 1010 . 0000 0000 . 0000 0000 . 0000 00	01	10.0.0.1
0000 1010 . 0000 0000 . 0000 0000 . 0000 00	10	10.0.0.2
0000 1010 . 0000 0000 . 0000 0000 . 0000 00	11	10.0.0.3

As we can see, we have “stolen” 6 bits from the host portion of the address in order to create our /30 network. This leaves us with only 2 bits in the host portion for addressing which means that the address range of 10.0.0.0/30 is 10.0.0.0 - 10.0.0.3, or 4 addresses (2 to the 2nd power).

What if we went the opposite direction and created a /23?

network bits	**host bits**	**address**
0000 1010 . 0000 0000 . 0000 000	0 . 0000 0000	10.0.0.0
0000 1010 . 0000 0000 . 0000 000	0 . 0000 0001	10.0.0.1
0000 1010 . 0000 0000 . 0000 000	0 . 0000 0002	10.0.0.2

network bits	**host bits**	**address**
…	…	…
0000 1010 . 0000 0000 . 0000 000	1 . 1111 1110	10.0.1.254
0000 1010 . 0000 0000 . 0000 000	1 . 1111 1111	10.0.1.255

As we can see, we have “stolen” 1 bit from the network portion of the address in order to create our /23 network. This gives us 9 bits in the host portion for addressing which means that the address range of 10.0.0.0/23 is 10.0.0.0 - 10.0.1.255, or 512 addresses (2 to the 9th power).

Notice the pattern here:

- for every bit we steal from the host portion, we halve the address space
- for every bit we give to the host portion, we double the address space

This ties back to our exercises in binary counting. Everything operates on powers of 2.

Switching gears a bit, lets consider some simple sizing exercises.

1. How many /25 networks can I create from a /24? In order to create /25 we would need to steal 1 bit from the host portion and give it to the network portion (25-24 = 1). 2 to the power of 1 is 2, so we would get 2 /25 networks from a /24.
2. How many /30 networks can I create from a /24? Similar to the previous exercise, 30 - 24 = 6 so we need to steal 6 bits. 2 to the power of 6 is 64. So 64 /30 networks can be created from a single /24.
3. How many addresses fit within a /30? We have 2 bits in the host portion of the address (32-30 = 2), so 2 to the 2nd power is 4.
4. How many address fit within a /22? We have 10 bits in the host portion of the address (32-22 = 10), so 2 to the 10th power is 1024.

Lets consider a more complex problem.

If I have a /24 and I need to create 5 subnets from it, what would be the maximum size of these subnets? How many subnets would I get? What would be their network addresses?

Firstly, determine the number of bits we need to steal. Look at this incrementally:

- 1 bit = 2 networks. Not enough.
- 2 bits = 4 networks. Not enough.
- 3 bits = 8 networks. Meets our requirement of 5 subnets.

So we need to steal 3 bits from the host portion, which gives us a /27. We can now determine the size of these subnets since we know that there are 5 bits remaining in the host portion of each

subnet (32-27 = 5) and that 2 to the 5th power = 32. So we know that we will end up with 8 /27 subnets which can each hold 32 addresses.

How do we determine the network addresses for these subnets. Lets look at the binary breakout using 10.0.0.0/24 as our original network:

network bits	**stolen bits**	**host bits**	**nework address**
0000 1010 . 0000 0000 . 0000 0000	. 000	0 0000	10.0.0.0/27
0000 1010 . 0000 0000 . 0000 0000	. 001	0 0000	10.0.0.32/27
0000 1010 . 0000 0000 . 0000 0000	. 010	0 0000	10.0.0.64/27
0000 1010 . 0000 0000 . 0000 0000	. 011	0 0000	10.0.0.96/27
0000 1010 . 0000 0000 . 0000 0000	. 100	0 0000	10.0.0.128/27
0000 1010 . 0000 0000 . 0000 0000	. 101	0 0000	10.0.0.160/27
0000 1010 . 0000 0000 . 0000 0000	. 110	0 0000	10.0.0.192/27
0000 1010 . 0000 0000 . 0000 0000	. 111	0 0000	10.0.0.224/27

Notice that when determining the network addresses of our 8 /27 subnets, we have only incremented the bits which we stole from the host portion of our original /24. You may also notice that the result is that we are incrementing our network address by the size of each subnet, or by 32 in this case.
