Mastering Network Protocols

an internet.com Networking eBook

Sometimes the simplest elements of modern networking are the easiest to forget, whether you're talking about calculating a subnet or understanding performance bottlenecks at the protocol level.

This guide is written with two audiences in mind: Network technicians who could use a refresher course on concepts they haven't needed since they crammed for their first certification, and IT managers with a background in disciplines outside networking who could use a cheat-sheet when they're dealing with this involved and complex field.

Whether we're covering new ground or introducing new concepts, we hope you find helpful this look at the foundation of networking.

Understanding IP Addresses

Networks don't work without addresses: Whenever you are sending something, you need to specify where it should go and where it came from. To be an effective network engineer or administrator, you need to understand IP addresses backward and forward: you need to be able to think on your feet. If something breaks, likely as not some address assignment has been screwed up. And spotting the problem quickly could be the difference between being the hero, or the person who "takes a long time to fix the problem." Thoroughly understanding IP addresses in their primal form is crucial to understanding subnets.

IPv4 Addresses and 32-bit Numbers

IP addresses are just 32-bit binary numbers, but they're important binary numbers: you need to how to work with them. When working with subnet masks, new network administrators generally get confused with the ones they haven't memorized. All the subnet mask amounts to is moving the boundary between the part of the address that represents a "network" and the part that represents a "host." Once you're comfortable with this method of thinking about IP addresses and masks, you've mastered IP addressing.

Binary is quite simple. In binary the only numbers are zeros and ones, and a 32-bit number holds 32 zeros and ones. We're all used to base-10 numbers, where each place in a number can hold any number from 0-9. In binary each place holds either a zero or a one. Here's the address 255.255.255.0 in binary:

11111111.11111111.11111111.00000000

For convenience, network engineers typically break IP addresses into four 8-bit blocks, or octets. In an 8-bit number, if all of the bits are set to 1, then the number is equal to 255. In the previous address, 11111111 represents 255 and 00000000 represents zero.

The way binary really works is based on powers of two. Each bit represents a different power-of-two. Starting at the left-hand side, the most significant bit, numbers form in the following manner:

Power-of-two	2^{32}	...	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}
Decimal	4,294,967,296	...	128	64	32	16	8	4	2	1

The result is additive, meaning that if all bits are set, you simply add the power-of-two value up for each place. For example, if we have an 8-bit number, 11111111 , we simply add: $2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 = 255$

Try a non-trivial example now: 11110000

We can see that four bits are "set" in the above 8-bit number. Summing the power-of-two values in those places yields: $2^7 + 2^6 + 2^5 + 2^4 = 240$

It's just that simple. If you can convert a binary number to decimal form, you can easily figure out subnet masks and network addresses.

Focusing on 32-bit IPv4 addresses themselves now, there are a few different types that need to be understood. All IP addresses can be in the range 0.0.0.0 to 255.255.255.255, but some have special uses.

Loopback
Packets that will not leave the host (i.e., they will not traverse an external network interface). Example: 127.0.0.1

Unicast
Packets that are destined for a single IP address. Example: 2.2.2.2

Multicast
Packets that will be duplicated by the router, and eventually routed by multicast routing mechanisms. Example: 226.0.0.2
Limited Broadcast

A broadcast packet, sent to every host, limited to the local subnet. Example: 255.255.255.255

Directed Broadcast
Packets that are routed to a specific subnet, and then broadcast. Example, assuming we are not on this subnet: 1.1.1.255

There are also some special cases of IP addresses, including private and multicast addresses. Addresses in the range 224.0.0.0 to 239.255.255.255 are reserved for multicast. Everything below that range is fair game on the Internet, excluding addresses reserved by RFC 1918 and a few other special-purpose assignments. These "1918 addresses" are private addresses, meaning Internet routers will not route them. The ranges include:

- 10.0.0.0 to 10.255.255.255
- 172.16.0.0 to 172.31.255.255
- 192.168.0.0 to 192.168.255.255

These IP addresses can be assigned locally to as many computers as you want, but before those computers access the Internet, the addresses must be translated to a globally routable address. This is commonly done via Network Address Translation (NAT). The 1918 addresses aren't the only reserved spaces, but they are defined to be "site local." Multicast also has a reserved range of addresses that aren't designed to escape onto the Internet: 224.0.0.0 to 224.0.0.255 are multicast "link-local" addresses.

The International Standards Organization (ISO) developed the OSI (Open Systems Interconnection) model. It divides network communication into seven layers. Layers 1-4 are considered the lower layers, and mostly concern themselves with moving data around. Layers 5-7, the upper layers, contain application-level data. Networks operate on one basic principle: "pass it on." Each layer takes care of a very specific job, and then passes the data onto the next layer.

Layer one is simply wiring, fiber, network cards, and anything else that is used to make two network devices communicate. Even a carrier pigeon would be considered layer one. Network troubleshooting will often lead to a layer one issue.

Layer two is Ethernet, among other protocols. The most important take-away from layer 2 is that you should understand what a bridge is. Switches, as they're called nowadays, are bridges. They all operate at layer 2, paying attention only to MAC addresses on Ethernet networks. If you're talking about MAC address, switches, or network cards and drivers, you're in the land of layer 2. Hubs live in layer 1 land.

If you're talking about an IP address, you're dealing with layer 3 and "packets" instead of layer 2's "frames." IP is part of layer 3, along with some routing protocols, and ARP (Address Resolution Protocol).

Layer four, the transport layer, handles messaging. Layer 4 data units are also called packets, but when you're talking about specific protocols, like TCP, they're "segments" or "datagrams" in UDP.

And arriving at layer 7, we wonder what happened to layer 5 and 6. In short: They're useless.

A few applications and protocols live there, but for understanding networking issues talking about these provides zero benefit. Layer 7, our friend, is "everything." Dubbed the "Application Layer," layer 7 is application-specific. If your program needs a specific format for data, you invent some format that you expect the data to arrive in and you've just created a layer 7 protocol. SMTP, DNS and FTP are all layer 7 protocols.

– Charlie Schluting, Enterprise Networking Planet

Let's review the concept of a local subnet. Once we have assigned a valid IP address to a computer, it will be able to speak to the local network, assuming the subnet mask is configured properly. The subnet mask tells the operating system which IP addresses are on the local subnet and which are not.

If an IP we wish to talk to is located on the local subnet, then the operating system can speak directly to it without using the router. In other words, it can ARP for the machine, and just start talking. IP address and subnet mask configuration is fairly straightforward for general /24 networks. The standard 255.255.255.0 mask means that the first three octets are the network address, and the last part is reserved for hosts. For example, a computer assigned the IP of 10.0.0.1 and a mask of 255.255.255.0 (a /24, or 24-bits if you write it out in binary) can talk to anyone inside the range 10.0.0.1 to 10.0.0.255.

Understanding Subnets and CIDR

Let's get one thing straight: there is no "Class" in subnetting. In the olden days, there were Class A, B, and C networks. These could only be divided up into equal parts, so VLSM (Variable Length Subnet Masks) were introduced. The old Class C was a /24, B was a /16, and A was a /8. That's all you need to know about Classes. They don't exist anymore.

An IP address consists of a host and a network portion. Coupled with a subnet mask, you can determine which part is the subnet, how large the network is, and where the network begins. Operating systems need to know this information in order to determine which IP addresses are on the local subnet and which addresses belong to the outside world and require a router to reach. Neighboring routers also need to know how large the subnet is, so they can send only applicable traffic that direction. Divisions between host and network portions of an address are completely determined by the subnet mask.

Classless Internet Domain Routing (CIDR), pronounced "cider," represents addresses using the network/mask style. What this really means is that an IP address/mask combo tells you a lot of information:

network part / host part
0000000000000000/0000000000000000

The above string of 32 bits represents a /16 network, since 16 bits are masked.

Throughout these examples (and in the real world), certain subnet masks are referred to repeatedly. They are not special in any way; subnetting is a simple string of 32 bits, masked by any number of bits. It is, however, helpful for memorizing and visualizing things to start with a commonly used netmask, like the /24, and work from there.

Let's take a look at a standard subnetting table, with a little bit different information:

Subnet mask bits	Number of /24 subnets	Number of addresses	Bits stolen
/24	1	256	0
/25	2	128	1
/26	4	64	2
/27	8	32	3
/28	16	16	4
/29	32	8	5
/30	64	4	6
/31	128	2	7

Because of the wonders of binary, it works out that a /31 has two IP addresses available. Imagine the subnet: 2.2.2.0/31. If we picture that in binary, it looks like:

00000010.00000010.00000010.00000000 (2.2.2.0)

11111111.11111111.11111111.11111110 (31)

The mask is "masking" the used bits, meaning that the bits are used up for network identification. The number of host bits available for tweaking is equal to one. It can be a 0 or a 1. This results in two available IP addresses, just like the table shows. Also, for each additional bit used in the netmask (stolen from the network portion), you can see that the number of available addresses gets cut in half.

Let's figure out the broadcast address, network address, and netmask for 192.168.0.200/26. The netmask is simple: that's 255.255.255.192 (26 bits of mask means 6 bits for hosts, 2^6 is 64, and 255-64 is 192). You can find subnetting tables online that will list all of this information for you, but we're more interested in teaching people how to understand what's happening. The netmask tells you immediately that the only part of the address we need to worry about is the last byte: the broadcast address and network address will both start with 192.168.0.

Figuring out the last byte is a lot like subnetting a /24 network, but you don't even need to think about that if it doesn't help you. Each /26 network has 64 hosts. The networks run from .0 to .64, .65 to .128, .128 to .192, and from .192 to .256. Our address, 192.168.0.200/26, falls into the .192 to .256 netblock. So the network address is 192.168.0.192/26. And the broadcast address is even simpler: 192 is 11000000 in binary. Take the last six bits (the bits turned "off" by the netmask), turn them "on", and what do you get? 192.168.0.255. To see if you got this right, compute the network address and broadcast address for 192.168.0.44/26. (Network address: 192.168.0.0/26; broadcast 192.168.0.63).

It can be hard to visualize these things at first, and it helps to start with making a table. If you calculated that you wanted subnets with six hosts in each of them, (eight, including the network and broadcast address that can't be used) then you can start making the table. The following is 2.2.2.0/29, 2.2.2.8/29, 2.2.2.16/29 and the final subnet of 2.2.2.249/29.

Subnet Number	Network Address	First IP	Last IP	Broadcast Address
1	2.2.2.0	2.2.2.1	2.2.2.6	2.2.2.7
2	2.2.2.8	2.2.2.9	2.2.2.14	2.2.2.15
3	2.2.2.16	2.2.2.17	2.2.2.22	2.2.2.23
32	2.2.2.249	2.2.2.250	2.2.2.254	2.2.2.255

In reality, you're much more likely to stumble upon a network where there's three /26's and the final /26 is divided up into two /27's. Being able to create the above table mentally will make things much easier.

That's really all you need to know. It gets a little trickier with larger subnets in the /16 to /24 range, but the principal is the same. It's 32 bits and a mask. Do, however, realize that there are certain restrictions governing the use of subnets. We cannot allocate a /26 starting with 10.1.0.32. If we utter the IP/mask of 10.1.0.32/26 to most operating systems, they will just assume we meant 10.1.0.0/26. This is because the /26 space requires 64 addresses, and they must start at a natural bit boundary for the given mask. In the above table, what would 2.2.2.3/29 mean? It means you meant to say 2.2.2.0/29.

Those tricky ones do demand a quick example. Remember how the number of IP addresses in a subnet gets

halved when you take another bit from the network side to create a larger mask? The same concept works in reverse. If we have a /25 that holds 128 hosts, and steal a bit from the host (netmask) portion, we now have a /24 that holds 256. Google for a "subnet table" to see the relationship between netmasks and network sizes all at once. If a /16 holds 65536 addresses, a /17 holds half as many, and a /15 holds twice as many. Practice, practice, practice. That's what it takes to understand how this works. Don't forget, you can always fall back to counting bits.

Understanding TCP, the Protocol

TCP is used everywhere, and understanding how TCP operates enables network and systems administrators to properly troubleshoot network communication issues.

TCP is wonderfully complex, but don't worry: We aren't going to tell you to go read RFC 793. This is a gentle introduction, or demystification, if you will. You'll be familiar with enough terminology, you'll understand the components of the TCP header, and then we'll discuss "TCP in the wild," which will focus on examining some common issues with TCP, including window scaling problems, congestion, and of course, the mechanics of a TCP connection.

We sometimes hear people call it "the TCP/IP protocol suite," which means that they're talking about layers 1-4 plus 7, similar to how we presented layers. TCP lives at layer 4, along with its unreliable friend UDP. TCP stands for Transmission Control Protocol. When a packet is encapsulated, we have the IP header at layer 3, and immediately following is the TCP header, which becomes the "data" for the IP header. TCP includes its own jargon, just like everything else. There were Ethernet frames, IP datagrams, and now TCP segments. You can think of them all as packets, but be sure to use the correct terms when communicating with others.

While trying to think of other things people say about TCP, it seemed apropos to spend some time explaining the things people are trying to tell you. There's nothing worse than asking a guru a question, and getting a response like "Well, it's end-to- end." If you knew TCP you'd know what this meant, but then you wouldn't have asked the question in the first place. Let's see what we can do about that.

Yes, TCP is end-to-end. There is no concept of broadcast, or anything like it. To speak TCP with another computer, you must be connected, like a telephone call, so each end is prepared to talk. "Stream delivery" is also another phrase you'll hear. This simply means that TCP works with data streams, and out of order packets are OK. In fact, TCP is even OK with lost or corrupted packets; it will eventually get them again.

More likely you'll be hearing a programmer talking about streams, referring to the fact that it's hard to tell when data is actually going to be sent, and you can send unstructured data down a TCP stream. TCP can buffer things, in weird ways that sometimes don't make sense, but neither programmers nor users need to worry about that.

Whenever a TCP packet is sent, an acknowledgment, or ACK, is returned. This is really the only way to provide a reliable protocol: You must let the other side know if you have received things. Of course, people will want to improve on an inefficient system like this. Enter "piggybacking ACKs" into the picture. People call TCP "full duplex" because of piggybacking; because it lets both sides send data at the same time. This is accomplished by carrying the ACK for previous packet received within the current packet, piggybacked. In terms of preserving network utilization, this is much better than sending an entirely separate packet just to say "got it." Finally, there's the concept of a cumulative ACK: ACKing more than one packet at a time, to say "I got all the others, including this one."

TCP is a stream, so there isn't really any other concept to worry about aside from a "connection." Maximum Segment Size, or MSS, is negotiated at connection time, but almost always changes. The default MSS is 536, which is 576 (the IP guaranteed minimum packet size) minus 20 bytes for the IP header and 20 bytes for the TCP header. TCP tries to avoid causing IP-level fragmentation, so it will almost always start with 536.

The sexiest feature of TCP still remains the Sliding Window Protocol. The window is essentially the amount of un-

ACKed data that has been sent, and it can grow and shrink at will. This gets really interesting, and will be covered next time.

The header of a TCP packet is 20 bytes, just like an IP's. Both IP and TCP headers can get larger, if options are used. TCP does not include an IP address; it only needs to know about the port on which to connect. Don't let this confuse you though; TCP keeps track of end-to-end connections in a state table that includes IP addresses and ports. It's just that the header for TCP doesn't need the IP information, since it comes from IP.

It is easier to think of a packet as a stream, one byte after the next. Everyone always wants to show a table for the header, but this can confuse matters more. The TCP header, starting with the first bit, is:

- Source port, 16 bits: my local TCP port that's used for this connection
- Destination port, 16 bits: The remote machine's TCP port that I'm talking to
- Sequence number, 32 bits: the number used to keep track of packet ordering
- Acknowledgment number, 32 bits: the previously received sequence number that we're ACKing
- Header length, 4 bits: the number of 32-bit words in the header. This is set to five, if no options are used
- Reserved, 6 bits: reserved for future use
- Flags, 6 bits total, each flag is one bit (on or off):
    - URG: urgent field pointer
    - ACK: this packet is (or includes) an ACK
    - PSH: push function (not used)
    - RST: reset, or terminate the connection
    - SYN: synchronization packet, aka Start Connection
    - FIN: final packet, start hang-up sequence
- Window size, 16 bits: begins with the ACK field that the receiving side will accept
- Checksum, 16 bits: a checksum of the TCP header and data
- Urgent pointer, 16 bits: an offset from the sequence number that points to the data following URG data
- Options: MSS, Window scale, and more. This is mostly the focus of our next installment on TCP.

Each side of the TCP connection uses the two pairs of IP address and Port to identify the connection, and sends the data on to the application that is listening on the port.

Let's go ahead and take a look at TCP operational issues, now that we know a little about what TCP actually is.

We said that TCP gets "connected" before any data can be sent. To make that work, the side that initiates a TCP connection will send a SYN (remember the Flags field) packet first. This is simply a packet with no data, and the SYN flag turned on. If the other side wants to talk on the port it received the SYN on, it will send back a SYN+ACK: SYN and ACK fields set, and the ACK number set to acknowledge the first packet. Then, to verify the receipt of the SYN+ACK, the sender will send one final ACK. The SYN, SYN+ACK, ACK sequence is called the three-way handshake. After that happens, the connection is established. The connection will remain active unless it times out or until either side sends a FIN.

Closing a TCP connection can be done from either side, and requires that both sides send a FIN to close their channel of communication. One side can close before the other, or they can both happen at the same time. When one side sends a FIN, the other sends FIN+ACK, to start the close of its side, and to ACK the first FIN. The person who sent the first FIN will then FIN+ACK the second FIN, and the other person knows that the connection is closed. There is no way for the person who sent the first FIN to get an ACK back for that last ACK. You might want to reread that now. The person that initially closed the connection enters the TIME_WAIT state; in case the other person didn't really get the ACK and thinks the connection is still open. Typically, this lasts one to two minutes.
And we've come to our first problem. If someone, say an attacker, leaves half-open or half-closed connections on your Web server, this could be bad news. Memory is used up with each connection, and opening thousands of

bogus TCP connections could bring a server to its knees. Of course, you can't really adjust the TCP timers without effecting the proper operation of TCP. If you've ever heard of a TCP SYN attack, this is what it means. To prevent this, most operating systems opt to limit the number of half-open connections, for example in Linux it's normally 256 by default.

Now, since we promised to talk about the everlasting flow control problems, let's get into windowing. TCP uses "positive ACK with retransmission" to guarantee reliability. The sender will wait a certain amount of time, and if it doesn't get back an ACK for the packet it sent, it retransmits it. There are many, many timers in TCP, by the way, this is just another one. The concept of ACKs is important to flow control, because the TCP sliding window protocol makes the ping-pong nature of ACKs efficient. If TCP were to send a packet and wait for every ACK, it would essentially cut the throughput in half.

Ideally, we can send many packets at once, and then get back an ACK for all of them, probably piggybacked on more data from the other side. But how do we know how much to send? Well, the TCP window size controls how many packets can be held in the "sent but not ACKed" state. If the window is large, we can send large amounts of packets without waiting for an ACK. On the surface, this doesn't look like flow control, but it certainly is.

The receiving side is the one that controls the window size. If it says zero, then the sender cannot send any more data at all. If the window size is one, then we're back to the simple "send and wait for ACK" protocol. If the last window size was zero, the sender will send a probe to figure out when the window is open again. If the sender never gets an ACK, it just keeps trying until a timer expires. Remember, the window size is a 16-bit field in the TCP header. And if you want a window size (in bytes) larger than 16-bits will accommodate, there's also a TCP option "window scale" that allows it to be multiplied by the scale factor. Without an extremely large window size, TCP has no hope of filling a gigabit link. You should now be better prepared to understand the gigabit-tuning article, too.

On the subject of TCP flow control, we can't neglect mention of the Nagle algorithm. What would happen if you had a large TCP window over a telnet connection? You'd type a command, then wait and wait and wait for a response. This is a major problem for real-time applications. Furthermore, telnet can add to congestion, since a 1-byte packet will include 40 bytes of header. RFC 896 defines the Nagle algorithm to attempt to abolish tiny packets. The idea is that we should give data a chance to pile up before sending, to be more efficient. It says that we can only have one unacked small segment, and you can't send more data until you get an ACK. Telnet and interactive ssh connections turn this off with the TCP_NODELAY socket option, so that you can get an immediate response when you press a key.