destroyallsoftware.com

Network Protocols – Programmer's Compendium

22-28 minutes

The network stack does several seemingly-impossible things. It does reliable transmission over our unreliable networks, usually without any detectable hiccups. It adapts smoothly to network congestion. It provides addressing to billions of active nodes. It routes packets around damaged network infrastructure, reassembling them in the correct order on the other side even if they arrived out of order. It accommodates esoteric analog hardware needs, like balancing the charge on the two ends of an Ethernet cable. This all works so well that users never hear of it, and even most programmers don't know how it works.

In the old days of analog telephones, making a phone call meant a continuous electrical connection from your phone to your friend's. It was as if a wire were run directly from you to them. There was no such wire, of course – the connection was made through complex switching systems – but it was electrically equivalent to a single wire.

There are too many Internet nodes for it to work in this way. We can't provide a direct, uninterruptible path from each machine to each other machine it wants to talk to.

Instead, data is bucket-brigaded – handed off from one router to the next, in a chain, each one bringing it closer to its destination. Each router between my laptop and google.com is connected to a number of other routers, maintaining a crude routing table showing which routers are closer to which parts of the Internet. When a packet arrives destined for google.com, a quick lookup in the routing table tells the router where the packet should go next to bring it closer to Google. The packets are small, so each router in the chain ties up the next router for only a tiny fraction of a second.

Routing breaks down into two sub-problems. First, addressing: what is the data's destination? This is handled by IP, the Internet Protocol, whence IP addresses. IPv4, still the most common version of IP, provides only 32 bits of address space. It's now fully allocated, so adding a node to the public Internet requires reusing an existing IP address. IPv6 allows 2^{128} addresses (about 10^{38}), but only has about 20% adoption as of 2017.

Now that we have addresses, we need to know how to route a packet through the Internet toward its destination. Routing happens fast, so there's no time to query remote databases for routing information. As an example, Cisco ASR 9922 routers have a maximum capacity of 160 terabits per second. Assuming full 1,500 byte packets (12,000 bits), that's 13,333,333,333 packets per second in a single 19 inch rack!

To route quickly, routers maintain *routing tables* indicating the paths to various groups of IP addresses. When a new packet arrives, the router looks it up in the table, telling it which peer is closest to the destination. It sends the packet to that peer and moves on to the next. BGP's job is to communicate this routing table information between different routers, ensuring up-to-date route tables.

IP and BGP together don't make a useful Internet, unfortunately, because they provide no way to transfer data reliably. If a router becomes overloaded and drops a packet, we need a way to detect the loss and request retransmission.

If the Internet works by routers handing data to each other down the line, what happens when the data is large? What if we request the 88.5 MB video of The Birth & Death of JavaScript?

We could try to design a network where the 88.5 MB document is sent from the web server to the first router, then to the second, and so on. Unfortunately, that network wouldn't work at Internet scale, or even at intranet scale.

First, computers are finite machines with finite amounts of storage. If a given router has only 88.4 MB of buffer memory available, it simply can't store the 88.5 MB video file. The data will be dropped on the floor and, worse, I'll get no indication. If a router is so busy that it's dropping data, it can't take the time to tell me about dropped data.

Second, computers are unreliable. Sometimes, routing nodes fail. Sometimes, ships' anchors accidentally damage underwater fiber-optic cables, taking out large portions of the Internet.

For these reasons and more, we don't send 88.5 MB messages across the Internet. Instead, we break them down into packets, usually in the neighborhood of 1,400 bytes each. Our video file will be broken into 63,214 or so separate packets for transmission.

Measuring an actual transfer of The Birth & Death of JavaScript with the packet capture tool Wireshark, I see a total of 61,807 packets received, each 1,432 bytes. Multiplying those two, we get 88.5 megabytes, which is the size of the video. (This doesn't include the overhead added by various protocols; if it did, we'd see a slightly higher number.)

The transfer was done over HTTP, a protocol layered over TCP, the Transmission Control Protocol. It only took 14 seconds, so the packets arrived at an average rate of about 4,400 per second, or about 250 microseconds per packet. In 14 seconds, my machine received all 61,807 of those packets, possibly out-of-order, reassembling them into the full file as they came in.

TCP packet reassembly is done using the simplest imaginable mechanism: a counter. Each packet is assigned a *sequence number* when it's sent. On the receiving side, the packets are put in order by sequence number. Once they're all in order, with no gaps, we know the whole file is present.

(Actual TCP sequence numbers tend not to be integers simply increasing by 1 each time, but that detail isn't important here.)

How do we know when the file is finished, though? TCP doesn't say anything about that; it's the job of higher-level protocols. For example, HTTP responses contain a "Content-Length" header specifying the response length in bytes. The client reads the Content-Length, then keeps reading TCP packets, assembling them back into their original order, until it has all of the bytes specified by Content-Length. This is one reason that HTTP headers (and most other protocols' headers) come before the response payload: otherwise, we wouldn't know the payload's size.

When we say "the client" here, we're really talking about the entire receiving computer. TCP reassembly happens inside the kernel, so applications like web browsers and curl and wget don't have to manually reassemble TCP packets. But the kernel doesn't handle HTTP, so applications do have to understand the Content-Length header and know how many bytes to read.

With sequence numbers and packet reordering, we can transmit large sequences of bytes even if the packets arrive out-of-order. But what if a packet is lost in transit, leaving a hole in the HTTP response?

I did a normal download of The Birth & Death of JavaScript with Wireshark turned on. Scrolling through the capture, I see packet after packet being received successfully.

For example, a packet with sequence number 563,321 arrived. Like all TCP packets, it had a "next sequence number", which is the number used for the following packet. This packet's "next sequence number" was 564,753. The next packet did, in fact, have sequence number 564,753, so everything was good. This happens thousands of times per second once the connection gets up to speed.

Occasionally, my computer sends a message to the server saying, for example, "I've received all packets up to and including packet number 564,753." That's an ACK, for *acknowledgement*: my computer acknowledges receipt of the server's packets. On a new connection, the Linux kernel sends an ACK after every ten packets. This is controlled by the `TCP_INIT_CWND` constant, which we can see defined in the Linux kernel's source code.

(The `CWND` in `TCP_INIT_CWND` stands for *congestion window*: the amount of data allowed in flight at once. If the network becomes *congested* – overloaded – then the window size will be reduced, slowing packet transmission.)

Ten packets is about 14 KB, so we're limited to 14 KB of data in flight at a time. This is part of *TCP slow start*: connections begin with small congestion windows. If no packets are lost, the receiver will continually increase the congestion window, allowing more packets in flight at once.

Eventually, a packet will be lost, so the receive window will be decreased, slowing transmission. By automatically adjusting the congestion window, as well as some other parameters, the sender and receiver keep data moving as quickly as the network will allow, but no quicker.

This happens on both sides of the connection: each side ACKs the other side's messages, and each side maintains its own congestion window. Asymmetric windows allow the protocol to take full advantage of network connections with asymmetric upstream and downstream bandwidth, like most residential and mobile Internet connections.

Computers are unreliable; networks made of computers are extra unreliable. On a large-scale network like the Internet, failure is a normal part of operation and must be accommodated. In a packet network, this means *retransmission*: if the client receives packets number 1 and 3, but doesn't receive 2, then it needs to ask the server to re-send the missing packet.

When receiving thousands of packets per second, as in our 88.5 MB video download, mistakes are almost guaranteed. To demonstrate that, let's return to my Wireshark capture of the download. For thousands of packets, everything goes normally. Each packet specifies a "next sequence number", followed by another packet with that number.

Suddenly, something goes wrong. The 6,269th packet has a "next sequence number" of 7,208,745, but that packet never comes. Instead, a packet with sequence number 7,211,609 arrives. This is an out-of-order packet: something is missing.

We can't tell exactly what went wrong here. Maybe one of the intermediate routers on the Internet was overloaded. Maybe my local router was overloaded. Maybe someone turned a microwave on, introducing electromagnetic interference and slowing my wireless connection. In any case, the packet was lost and the only indication is the unexpected packet.

TCP has no special "I lost a packet!" message. Instead, ACKs are cleverly reused to indicate loss. Any out-of-order packet causes the receiver to re-ACK the last "good" packet – the last one in the correct order. In effect, the receiver is saying "I received packet 5, which I'm ACKing. I also received something after that, but I know it wasn't packet 6 because it didn't match the next sequence number in packet 5."

If two packets simply got switched in transit, this will result in a single extra ACK and everything will continue normally after the out-of-order packet is received. But if the packet was truly lost, unexpected packets will continue to arrive and the receiver will continue to send duplicate ACKs of the last good packet. This can result in hundreds of duplicate ACKs.

When the sender sees three duplicate ACKs in a row, it assumes that the following packet was lost and retransmits it. This is called *TCP fast retransmit* because it's faster than the older, timeout-based approach. It's interesting to note that the protocol itself doesn't have any explicit way to say "please retransmit this immediately!" Instead, multiple ACKs arising naturally from the protocol serve as the trigger.

(An interesting thought experiment: what happens if some of the duplicate ACKs are lost, never reaching the sender?)

Retransmission is common even in networks working normally. In a capture of our 88.5 MB video download, I saw this:

- The congestion window quickly increases to about a megabyte due to continuing successful transmission.
- A few thousand packets show up in order; everything is normal.
- One packet comes out of order.
- Data continues pouring in at megabytes per second, but the packet is still missing.

- My machine sends dozens of duplicate ACKs of the last known-good packet, but the kernel also stores the pending out-of-order packets for later reassembly.
- The server receives the duplicate ACKs and resends the missing packet.
- My client ACKs both the previously-missing packet and the later ones that were already received due to out-of-order transmission. This is done by simply ACKing the most recent packet, which implicitly ACKs all earlier ones as well.
- The transfer continues, but with a reduced congestion window due to the lost packet.

This is normal; it's happened in every capture of the full download that I've done. TCP is so successful at its job that we don't even think of networks as being unreliable in our daily use, even though they fail routinely under normal conditions.

All of this network data has to be transferred over physical media like copper, fiber optics, and wireless radio. Of the physical layer protocols, Ethernet is the most well known. Its popularity in the early days of the Internet led us to design other protocols to accommodate its limitations.

First, let's get the physical details out of the way. Ethernet is most closely associated with RJ45 connectors, which look like bigger eight-pin versions of older four-pin phone jacks. It's also associated with cat5 (or cat5e, or cat6, or cat7) cable, which contains eight total wires twisted into four pairs. Other media exist, but these are the ones we're most likely to encounter at home: eight wires wrapped in a sheath connected to an eight-pin jack.

Ethernet is a *physical layer protocol*: it describes how the bits turn into electrical signals in a cable. It's also a *link layer protocol*: it describes the direct connection of one node to another. However, it's purely point-to-point and says nothing about how data is routed on a network. There's no concept of a connection in the sense of a TCP connection, or of reassignable addresses in the sense of an IP address.

As a protocol, ethernet has two primary jobs. First, each device needs to notice that it's connected to something, and some parameters like connection speed need to be negotiated.

Second, once link is established, Ethernet needs to carry data. Like the higher-level protocols TCP and IP, Ethernet data is broken into packets. The core of a packet is a *frame*, which has a 1,500 byte payload, plus another 22 bytes for header information like source and destination MAC address, payload length, and checksum. These fields are familiar: programmers often deal with addresses and lengths and checksums, and we can imagine why they're necessary.

The frame is then wrapped in yet another layer of headers to form the full packet. These headers are... weird. They start to bump up against the underlying reality of analog electrical systems, so they look like nothing we would ever put in a software protocol. A full Ethernet packet contains:

- The preamble, which is 56 bits (7 bytes) of alternating 1s and 0s. The devices use this to synchronize their clocks, sort of like when people count off "1-2-3-GO!" Computers can't count past 1, so they synchronize by saying "10101010101010101010101010101010101010101010101010101010".

- An 8-bit (1 byte) start frame delimiter, which is the number 171 (10101011 in binary). This marks the end of the preamble. Notice that it's "10" repeated again, until the end where there's a "11".
- The frame itself, which contains the source and destination addresses, the payload, etc., as described above.
- An interpacket gap of 96 bits (12 bytes) where the line is left idle. Presumably, this is to let the devices rest because they are tired.

Putting this all together: what we want is to transmit our 1,500 bytes of data. We add 22 bytes to create a frame, which indicates the source, destination, size, and checksum. We add another 20 bytes of extra data accommodating the hardware's needs, creating a full Ethernet packet.

You might think this is the bottom of the stack. It's not, but things do get weirder because the analog world pokes through even more.

Digital systems don't exist; everything is analog.

Suppose we have a 5-volt CMOS system. (CMOS is a type of digital system; don't worry about it if you're not familiar.) This means that a fully-on signal will be 5 volts, and a fully-off signal will be 0. But nothing is ever fully on or fully off; the physical world doesn't work like that. In reality, our 5-volt CMOS system will consider anything above 1.67 volts to be a 1, and anything below 1.67 to be 0.

(1.67 is 1/3 of 5. Let's not worry about why the threshold is 1/3. If you want to dig, there's a wikipedia article, of course! Also, Ethernet isn't CMOS or even related to CMOS, but CMOS and its 1/3 cutoff make for a simple illustration.)

Our Ethernet packets have to go over a physical wire, which means changing the voltage across the wire. Ethernet is a 5-volt system, so we naively expect each 1 bit in the Ethernet protocol to be 5 volts and each 0 bit to be 0 volts. But there are two wrinkles: first, the voltage range is -2.5 V to +2.5 V. Second, and more strangely, each set of 8 bits gets expanded into 10 bits before hitting the wire.

There are 256 possible 8-bit values and 1024 possible 10-bit values, so imagine this as a table mapping them. Each 8-bit byte can be mapped to any of four different 10-bit patterns, each of which will be turned back into the same 8-bit byte on the receiving end. For example, the 10-bit value 00.0000.0000 might map to the 8-bit value 0000.0000. But maybe the 10-bit value 10.1010.1010 also maps to 0000.0000. When an Ethernet device sees either 00.0000.0000 or 10.1010.1010, they'll be understood as the byte 0 (binary 0000.0000).

(Warning: there are going to be some electronics words now.)

This exists to serve an extremely analog need: balancing the voltage in the devices. Suppose this 8-bit-to-10-bit encoding doesn't exist, and we send some data that happens to be all 1s. Ethernet's voltage range is -2.5 to +2.5 volts, so we're holding the Ethernet cable's voltage at +2.5 V, continually pulling electrons from the other side.

Why do we care about one side pulling more electrons than the other? Because the analog world is a mess and it will cause all kinds of undesirable effects. To take one: it can charge the capacitors used in low-pass filters, creating an offset in the signal level itself, eventually causing bit errors. Those errors would take time to accumulate, but we don't want our network devices to suddenly corrupt data after two years of uptime simply because we happened to send more binary 1s than 0s.

(Electronics words end here.)

By using an 8b/10b encoding, Ethernet can balance the number of 0s and 1s sent over the wire, even if we send data that's mostly 1s or mostly 0s. The hardware tracks the ratio of 0s to 1s, mapping outgoing 8-bit bytes to different options from the 10-bit table to achieve electrical balance. (Newer Ethernet standards, like 10 GB Ethernet, use different and more complex encoding systems.)

We'll stop here, because we're already beyond the scope of what can be considered programming, but there are many more protocol issues to accommodate the physical layer. In many cases, the solutions to hardware problems lie in the software itself, as in the case of the 8b/10b coding used to correct DC offset. This is perhaps a bit disconcerting to us as programmers: we like to pretend that our software lives in a perfect Platonic world, devoid of the vulgar imperfections of physicality. In reality, everything is analog, and accommodating that complexity is everyone's job, including the software's.

Internet protocols are best thought of as a stack of layers.

Ethernet provides physical data transfer and link between two point-to-point devices. IP provides a layer of addressing, allowing routers and large-scale networks to exist, but it's connectionless. Packets are fired into the ether, with no indication of whether they arrived or not. TCP adds a layer of reliable transmission by using sequence numbers, acknowledgement, and retransmission.

Finally, application-level protocols like HTTP are layered on top of TCP. At this level, we already have addressing and the illusion of reliable transmission and persistent connections. IP and TCP save application developers from constantly reimplementing packet retransmission and addressing and so on.

The independence of these layers is important. For example, when packets were lost during my 88.5 MB video transfer, the Internet's backbone routers didn't know; only my machine and the web server knew. Dozens of duplicate ACKs from my computer were all dutifully routed over the same routing infrastructure that lost the original packet. It's possible that the router responsible for dropping the lost packet was also the router carrying its replacement milliseconds later. This is an important point for understanding the Internet: the routing infrastructure doesn't know about TCP; it only routes. (There are exceptions to this, as always, but it's generally true.)

Layers of the protocol stack operate independently, but they weren't designed independently. Higher-level protocols tend to be built on lower-level ones: HTTP is built on TCP is built on IP is built on Ethernet. Design decisions in lower levels often influence decisions in higher levels, even decades later.

Ethernet is old and concerns the physical layer, so its needs set the base parameters. An Ethernet payload is at most 1,500 bytes.

The IP packet needs to fit within an Ethernet frame. IP has a minimum header size of 20 bytes, so the maximum payload of an IP packet is 1,500 - 20 = 1,480 bytes.

Likewise, the TCP packet needs to fit within the IP packet. TCP also has a minimum header size of 20 bytes, leaving a maximum TCP payload of 1,480 - 20 = 1,460 bytes. In practice, other headers and protocols can cause further reductions. 1,400 is a conservative TCP payload size.

The 1,400 byte limit influences modern protocols' designs. For example, HTTP requests are generally small. If we fit them into one packet instead of two, we reduce the probability of losing part of the request, with a correspondingly reduced likelihood of TCP retransmissions. To squeeze every byte out of small requests, HTTP/2 specifies compression for headers, which are usually small. Without context from TCP, IP, and Ethernet, this seems silly: why add compression to a protocol's headers to save only a few bytes? Because, as the HTTP/2 spec says in the introduction to section 2, compression allows "many requests to be compressed into one packet".

HTTP/2 does header compression to meet the constraints of TCP, which come from constraints in IP, which come from constraints in Ethernet, which was developed in the 1970s, introduced commercially in 1980, and standardized in 1983.

One final question: why is the Ethernet payload size set at 1,500 bytes? There's no deep reason; it's just a nice trade-off point. There are 42 bytes of non-payload data needed for each frame. If the payload maximum were only 100 bytes, only 70% (100/142) of time would be spent sending payload. A payload of 1,500 bytes means about 97% (1500/1542) of time is spent sending payload, which is a nice level of efficiency. Pushing the packet size higher would require larger buffers in the devices, which we can't justify simply to get another percent or two of efficiency. In short: HTTP/2 has header compression because of the RAM limitations of networking devices in the late 1970s.