apenwarr
Je me souviens

Everything here is my opinion. I do not speak for your employer.

← *August 2017* *September 2017* →

2017-08-10 »

The world in which IPv6 was a good design

Last November I went to an IETF meeting for the first time. The IETF is an interesting place; it seems to be about 1/3 maintenance grunt work, 1/3 extending existing stuff, and 1/3 blue sky insanity. I attended mostly because I wanted to see how people would react to TCP BBR, which was being presented there for the first time. (Answer: mostly positively, but with suspicion. It kinda seemed too good to be true.)

Anyway, the IETF meetings contain lots and lots of presentations about IPv6, the thing that was supposed to replace IPv4, which is what the Internet runs on. (Some would say IPv4 is already being replaced; some would say it has already happened.) Along with those presentations about IPv6, there were lots of people who think it's great, the greatest thing ever, and they're pretty sure it will finally catch on Any Day Now, and IPv4 is just a giant pile of hacks that really needs to die so that the Internet can be elegant again.

I thought this would be a great chance to really try to figure out what was going on. Why is IPv6 such a complicated mess compared to IPv4? Wouldn't it be better if it had just been IPv4 with more address bits? But it's not, oh goodness, is it ever not. So I started asking around. Here's what I found.

Buses ruined everything

Once upon a time, there was the telephone network, which used physical circuit switching. Essentially, that meant moving connectors around so that your phone connection was literally just a very long wire ("OSI layer 1"). A "leased line" was a very long wire that you leased from the phone company. You would put bits in one end of the wire, and they'd come out the other end, a fixed amount of time later. You didn't need addresses because there was exactly one machine at each end.

Eventually the phone company optimized that a bit. Time-division multiplexing (TDM) and "virtual circuit switching" was born. The phone company could transparently take the bits at a slower bit rate from multiple lines, group them together with multiplexers and demultiplexers, and let them pass through the middle of the phone system using fewer wires than before. Making that work was a little complicated, but as far as we modem users were concerned, you still put bits in one end and they came out the other end. No addresses needed.

The Internet (not called the Internet at the time) was built on top of these circuits. You had a bunch of wires that you could put bits into and have them come out the other side. If one computer had two or three interfaces, then it could, if given the right instructions, forward bits from one line to another, and you could do something a lot more efficient than a separate line between each pair of computers. And so IP addresses ("layer 3"), subnets, and routing were born. Even then, with these point-to-point links, you didn't need MAC addresses, because once a packet went into the wire, there was only one place it could come out. You used IP addresses to decide where it should go after that.

Meanwhile, LANs got invented as an alternative. If you wanted to connect computers (or terminals and a mainframe) together at your local site, it was pretty inconvenient to need multiple interfaces, one for each wire to each satellite computer, arranged in a star configuration. To save on electronics, people wanted to have a "bus" network (also known as a "broadcast domain," a name that will be important later) where multiple stations could just be plugged into a single wire, and talk to any other station plugged into the same wire. These were not the same people as the ones building the Internet, so they didn't use IP addresses for this. They all invented their own scheme ("layer 2").

One of the early local bus networks was arcnet, which is dear to my heart (I wrote the first Linux arcnet driver and arcnet poetry way back in the 1990s, long after arcnet was obsolete). Arcnet layer 2 addresses were very simplistic: just 8 bits, set by jumpers or DIP switches on the back of the network card. As the network owner, it was your job to configure the addresses and make sure you didn't have any duplicates, or all heck would ensue. This was kind of a pain, but arcnet networks were usually pretty small, so it was only *kind* of a pain.

A few years later, ethernet came along and solved that problem once and for all, by using many more bits (48, in fact) in the layer 2 address. That's enough bits that you can assign a different (sharded-sequential) address to every device that has ever been manufactured, and not have any overlaps. And that's exactly what they did! Thus the ethernet MAC address was born.

Various LAN technologies came and went, including one of my favourites, IPX (Internetwork Packet Exchange, though it had nothing to do with the "real" Internet) and Netware, which worked great as long as all the clients and servers were on a single bus network. You never had to configure any addresses, ever. It was beautiful, and reliable, and worked. The golden age of networking, basically.

Of course, someone had to ruin it: big company/university networks. They wanted to have so many computers that sharing 10 Mbps of a single bus network between them all became a huge bottleneck, so they needed a way to have multiple buses, and then interconnect - "internetwork," if you will - those buses together. You're probably thinking, of course! Use the Internet Protocol for that, right? Ha ha, no. The Internet protocol, still not called that, wasn't mature or popular back then, and nobody took it seriously. Netware-over-IPX (and the many other LAN protocols at the time) were serious business, so as serious businesses do, they invented their own thing(s) to extend the already-popular thing, ethernet. Devices on ethernet already had addresses, MAC addresses, which were about the only thing the various LAN protocol people could agree on, so they decided to use ethernet addresses as the keys for their routing mechanisms. (Actually they called it bridging and switching instead of routing.)

The problem with ethernet addresses is they're assigned sequentially at the factory, so they can't be hierarchical. That means the "bridging table" is not as nice as a modern IP routing table, which can talk about the route for a whole subnet at a time. In order to do efficient bridging, you had to remember which network bus each MAC address could be found on. And humans didn't want to configure each of those by hand, so it needed to figure itself out automatically. If you had a complex internetwork of bridges, this could get a little complicated. As I understand it, that's what led to the spanning tree poem, and I think I'll just leave it at that. Poetry is very important in networking.

Anyway, it mostly worked, but it was a bit of a mess, and you got broadcast floods every now and then, and the routes weren't always optimal, and it was pretty much impossible to debug. (You definitely couldn't write something like traceroute for bridging, because none of the tools you need to make it work - such as the ability for an intermediate bridge to even have an address - exist in plain ethernet.)

On the other hand, all these bridges were hardware-optimized. The whole system was invented by hardware people, basically, as a way of fooling the software, which had no idea about multiple buses and bridging between them, into working better on large networks. Hardware bridging means the bridging could go really really fast - as fast as the ethernet could go. Nowadays that doesn't sound very special, but at the time, it was a big deal. Ethernet was 10 Mbps, because you could maybe saturate it by putting a bunch of computers on the network all at once, not because any one computer could saturate 10 Mbps. That was crazy talk.

Anyway, the point is, bridging was a mess, and impossible to debug, but it was fast.

Internet over buses

While all that was happening, those Internet people were getting busy, and were of course not blind to the invention of cool cheap LAN technologies. I think it might have been around this time that the ARPANET got actually renamed to the Internet, but I'm not sure. Let's say it was, because the story is better if I sound confident.

At some point, things progressed from connecting individual Internet computers over point-to-point long distance links, to the desire to connect whole LANs together, over point-to-point links. Basically, you wanted a long-distance bridge.

You might be thinking, hey, no big deal, why not just build a long distance bridge and be done with it? Sounds good, doesn't work. I won't go into the details right now, but basically the problem is congestion control. The deep dark secret of ethernet bridging is that it assumes all your links are about the same speed, and/or completely uncongested, because they have no way to slow down. You just blast data as fast as you can, and expect it to arrive. But when your ethernet is 10 Mbps and your point-to-point link is 0.128

Mbps, that's completely hopeless. Separately, the idea of figuring out your routes by flooding all the links to see which one is right - this is the actual way bridging typically works - is hugely wasteful for slow links. And sub-optimal routing, an annoyance on local networks with low latency and high throughput, is nasty on slow, expensive long-distance links. It just doesn't scale.

Luckily, those Internet people (if it was called the Internet yet) had been working on that exact set of problems. If we could just use Internet stuff to connect ethernet buses together, we'd be in great shape.

And so they designed a "frame format" for Internet packets over ethernet (and arcnet, for that matter, and every other kind of LAN).

And that's when everything started to go wrong.

The first problem that needed solving was that now, when you put an Internet packet onto a wire, it was no longer clear which machine was supposed to "hear" it and maybe forward it along. If multiple Internet routers were on the same ethernet segment, you couldn't have them all picking it up and trying to forward it; that way lies packet storms and routing loops. No, you had to choose *which* router on the ethernet bus is supposed to pick it up. We can't just use the IP destination field for that, because we're already using that for the *final* destination, not the router destination. Instead, we identify the desired router using its MAC address in the ethernet frame.

So basically, to set up your local IP routing table, you want to be able to say something like, "send packets to IP address 10.1.1.1 via the router at MAC address 11:22:33:44:55:66." That's the actual thing you want to express. This is important! Your destination is an IP address, but your router is a MAC address. But if you've ever configured a routing table, you might have noticed that nobody writes it like that. Instead, because the writers of your operating system's TCP/IP stack are stubborn, you write something like "send packets to IP address 10.1.1.1 via the router at IP address 192.168.1.1."

In truth, that really is just complicating things. Now your operating system has to first look up the ethernet address of 192.168.1.1, find out it's 11:22:33:44:55:66, and finally generate a packet with destination ethernet address 11:22:33:44:55:66 and destination IP address 10.1.1.1. 192.168.1.1 shows up nowhere in the packet; it's just an abstraction at the human level.

To do that pointless intermediate step, you need to add ARP (address resolution protocol), a simple non-IP protocol whose job it is to convert IP addresses to ethernet addresses. It does this by broadcasting to everyone on the local ethernet bus, asking them all to answer if they own that particular IP address. If you have bridges, they all have to forward all the ARP packets to all their interfaces, because they're ethernet broadcast packets, and that's what broadcasting means. On a big, busy ethernet with lots of interconnected LANs, excessive broadcasts start becoming one of your biggest nightmares. It's especially bad on wifi. As time went on, people started making bridges/switches with special hacks to avoid forwarding ARP as far as it's technically supposed to go, to try to cut down on this problem. Some devices (especially wifi access points) just make fake ARP answers to try to help. But doing any of that is a hack, albeit sometimes a necessary hack.

Death by legacy

Time passed. Eventually (and this actually took quite a while), people pretty much stopped using non-IP protocols on ethernet at all. So basically all networks became a physical wire (layer 1), with multiple stations on a bus (layer 2), with multiple buses connected over bridges (gotcha! still layer 2!), and those inter-buses connected over IP routers (layer 3).

After a while, people got tired of manually configuring IP addresses, arcnet style, and wanted them to auto-configure, ethernet style, except it was too late to literally do it ethernet style, because a) the devices had already been manufactured with ethernet addresses, not IP addresses, and b) IP addresses were only 32 bits, which is not enough to just manufacture them forever with no overlaps, and c) just assigning IP addresses sequentially instead of using subnets would bring us back to square one: it would just be ethernet over again, and we already have ethernet.

So that's where bootp and DHCP came from. Those protocols, by the way, are special kinda like ARP is special (except they pretend not to be special, by technically being IP packets). They have to be special, because an IP node has to be able to transmit them before it has an IP address, which is of course impossible, so it just fills the IP headers with essentially nonsense (albeit nonsense specified by an RFC), so the headers might as well have been left out. (You know these "IP" headers are nonsense because the DHCP server has to open a raw socket and fill them in by hand; the kernel IP layer can't do it.) But nobody would feel nice if they were inventing a whole new protocol that wasn't IP, so they pretended it was IP, and then they felt nice. Well, as nice as one can feel when one is inventing DHCP.

Anyway, I digress. The salient detail here is that unlike real IP services, bootp and DHCP need to know about ethernet addresses, because after all, it's their job to hear your ethernet address and assign you an IP address to go with it. They're basically the reverse of ARP, except we can't say that, because there's a protocol called RARP that is literally the reverse of ARP. Actually, RARP worked quite fine and did the same thing as bootp and DHCP while being much simpler, but we don't talk about that.

The point of all this is that ethernet and IP were getting further and further intertwined. They're nowadays almost inseparable. It's hard to imagine a network interface (except ppp0) without a 48-bit MAC address, and it's hard to imagine that network interface working without an IP address. You write your IP routing table using IP addresses, but of course you know you're lying when you name the router by IP address; you're just indirectly saying that you want to route via a MAC address. And you have ARP, which gets bridged but not really, and DHCP, which is an IP packet but is really an ethernet protocol, and so on.

Moreover, we still have both bridging and routing, and they both get more and more complicated as the LANs and the Internet get more and more complicated, respectively. Bridging is still, mostly, hardware based and defined by IEEE, the people who control the ethernet standards. Routing is still, mostly, software based and defined by the IETF, the people who control the Internet standards. Both groups still try to pretend the other group doesn't exist. Network operators basically choose bridging vs routing based on how fast they want it to go and how much they hate configuring DHCP servers, which they really hate very much, which means they use bridging as much as possible and routing when they have to.

In fact, bridging has gotten so completely out of control that people decided to extract the layer 2 bridging decisions out completely to a higher level (with configuration exchanged between bridges using a protocol layered over IP, of course!) so it can be centrally managed. That's called software-defined networking (SDN). It helps a lot, compared to letting your switches and bridges just do whatever they want, but it's also fundamentally silly, because you know what's software defined networking? IP. It is literally and has always been the software-defined network you use for interconnecting networks that have gotten too big. But the problem is, IPv4 was initially too hard to hardware accelerate, and anyway, it didn't get hardware accelerated, and configuring DHCP really is a huge pain, so network operators just learned how to bridge bigger and bigger things. And nowadays big data centers are basically just SDNed, and you might as well not be using IP in the data center at all, because nobody's routing the packets. It's all just one big virtual bus network.

It is, in short, a mess.

Now forget I said all that...

Great story, right? Right. Now pretend none of that happened, and we're back in the early 1990s, when most of that had in fact already happened, but people at the IETF were anyway pretending that it hadn't happened and that the "upcoming" disaster could all be avoided. This is the good part!

There's one thing I forgot to mention in that big long story above: somewhere in that whole chain of events, **we completely stopped using bus networks**. Ethernet is not actually a bus anymore. It just *pretends* to be a bus. Basically, we couldn't get ethernet's famous CSMA/CD to keep working as speeds increased, so we went back to the good old star topology. We run bundles of cables from the switch, so that we can run one cable from each station all the way back to the center point. Walls and ceilings and floors are filled with big, thick, expensive bundles of ethernet, because we couldn't figure out how to make buses work well... at layer 1. It's kinda funny actually when you think about it. If you find sad things funny.

In fact, in a bonus fit of insanity, even wifi - the ultimate bus network, right, where literally everybody is sharing the same open-air "bus" - we almost universally use wifi in a mode, called "infrastructure mode," which simulates a giant star topology. If you have two wifi stations connected to the same access point, they don't talk to each other directly, even when they can hear each other just fine. They send a packet to the access point, but addressed to the MAC address of the other node. The access point then bounces it back out to the destination node.

HOLD THE HORSES LET ME JUST REVIEW THAT FOR YOU. There's a little catch there. When node X wants to send to Internet node Z, via IP router Y, via wifi access point A, what does the packet look like? Just to draw a picture, here's what we want to happen:

X -> [wifi] -> A -> [wifi] -> Y -> [internet] -> Z

Z is the IP destination, so obviously the IP destination field has to be Z. Y is the router, which we learned above that we specify by using its ethernet MAC address in the ethernet destination field. But in wifi, X can't just send out a packet to Y, for various reasons (including that they don't know each other's WPA2 encryption keys). We have to send to A. Where do we put A's address, you might ask?

No problem! 802.11 has a thing called 3-address mode. They add a *third* ethernet MAC address to every frame, so they can talk about the real ethernet destination, and the intermediate ethernet destination. On top of that, there are bit fields called "to-AP" and "from-AP," which tell you if the packet is going from a station to an AP, or from an AP to a station, respectively. But actually they can both be true at the same time, because that's how you make wifi repeaters (APs send packets to APs).

Speaking of wifi repeaters! If A is a repeater, it has to send back to the base station, B, along the way, which looks like this:

X -> [wifi] -> A -> [wifi-repeater] -> B -> [wifi] -> Y -> [internet] -> Z

X->A uses three-address mode, but A->B has a problem: the ethernet source address is X, and the ethernet destination address is Y, but the packet on the air is actually being sent from A to B; X and Y aren't involved at all. Suffice it to say that there's a thing called 4-address mode, and it works pretty much like you think.

(In 802.11s mesh networks, there's a 6-address mode, and that's about where I gave up trying to understand.)

Avery, I was promised IPv6, and you haven't even mentioned IPv6

Oh, oops. This post went a bit off the rails, didn't it?

Here's the point of the whole thing. The IETF people, when they were thinking about IPv6, saw this mess getting made - and maybe predicted some of the additional mess that would happen, though I doubt they could have predicted SDN and wifi repeater modes - and they said, hey wait a minute, stop right there. We don't need any of this crap! What if instead the world worked like this?

- No more physical bus networks (already done!)
- No more layer 2 internetworks (that's what layer 3 is for)
- No more broadcasts (layer 2 is always point-to-point, so where would you send the broadcast to? replace it with multicast instead)
- No more MAC addresses (on a point-to-point network, it's obvious who the sender and receiver are, and you can do multicast using IP addresses)
- No more ARP and DHCP (no MAC addresses, no so mapping IP addresses to MAC addresses)
- No more complexity in IP headers (so you can hardware accelerate IP routing)
- No more IP address shortages (so we can go back to routing big subnets again)
- No more manual IP address configuration except at the core (and there are so many IP addresses that we can recursively hand out subnets down the tree from there)

Imagine that we lived in such a world: wifi repeaters would just be IPv6 routers. So would wifi access points. So would ethernet switches. So would SDN. ARP storms would be gone. "IGMP snooping bridges" would be gone. Bridging loops would be gone. Every routing problem would be traceroute-able. And best of all, we could drop 12 bytes (source/dest ethernet addresses) from every ethernet packet, and 18 bytes (source/dest/AP addresses) from every wifi packet. Sure, IPv6 adds an extra 24 bytes of address (vs IPv4), but you're dropping 12 bytes of ethernet, so the added overhead is only 12 bytes - pretty comparable to using two 64-bit IP addresses but having to keep the ethernet header. The idea that we could someday drop ethernet addresses helped to justify the oversized IPv6 addresses.

It would have been beautiful. Except for one problem: it never happened.

Requiem for a dream

One person at work put it best: "layers are only ever added, never removed."

All this wonderfulness depended on the ability to start over and throw away the legacy cruft we had built up. And that is, unfortunately, pretty much impossible. Even if IPv6 hits 99% penetration, that doesn't mean we'll be rid of IPv4. And if we're not rid of IPv4, we won't be rid of ethernet addresses, or wifi addresses. And if we have to keep the IEEE 802.3 and 802.11 framing standards, we're never going to save those bytes. So we will always need the "IPv6 neighbour discovery" protocol, which is just a more complicated ARP. Even though we no longer have bus networks, we'll always need some kind of simulator for broadcasts, because that's how ARP works. We'll need to keep running a local DHCP server at home so that our obsolete IPv4 light bulbs keep working. We'll keep needing NAT so that our obsolete IPv4 light bulbs can keep reaching the Internet.

And that's not the worst of it. The worst of it is we still need the infinite abomination that is layer 2 bridging, because of one more mistake the IPv6 team *forgot to fix*. Unfortunately, while they were blue-skying IPv6 back in the 1990s, they neglected to solve the "mobile IP" problem. As I understand it, the idea was to get IPv6 deployed first - it should only take a few years - and then work on it after IPv4 and MAC addresses had been eliminated, at which time it should be much easier to solve, and meanwhile, nobody really has a "mobile IP" device yet anyway. I mean, what would that even mean, like carrying your laptop around and plugging into a series of one ethernet port after another while you ftp a file? Sounds dumb.

The killer app: mobile IP

Of course, with a couple more decades of history behind us, now we know a few use cases for carrying around a computer - your phone - and letting it plug into one ~~ethernet port~~ wireless access point after another. We do it all the time. And with LTE, it even mostly works! With wifi, it works sometimes. Good, right?

Not really, because of the Internet's secret shame: all that stuff only works because of layer 2 bridging. Internet routing can't handle mobility - at all. If you move around on an IP network, your IP address changes, and that breaks any connections you have open.

Corporate wifi networks fake it for you, bridging their whole LAN together at layer 2, so that the giant central DHCP server always hands you the same IP address no matter which corporate wifi access point you join, and then gets your packets to you, with at most a few seconds of confusion while the bridge reconfigures. Those newfangled home wifi systems with multiple extenders/repeaters do the same trick. But if you switch from one wifi network to another as you walk down the street - like if there's a "Public Wifi" service in a series of stores - well, too bad. Each of those gives you a new IP address, and each time your IP address changes, you kill all your connections.

LTE tries even harder. You keep your IP address (usually an IPv6 address in the case of mobile networks), even if you travel miles and miles and hop between numerous cell towers. How? Well... they typically just tunnel all your traffic back to a central location, where it all gets bridged together (albeit with lots of firewalling) into one super-gigantic virtual layer 2 LAN. And your connections keep going. At the expense of a ton of complexity, and a truly embarrassing amount of extra latency, which they would really like to fix, but it's almost impossible.

Making mobile IP actually work[1]

So okay, this has been a long story, but I managed to extract it from those IETF people eventually. When we got to this point - the problem of mobile IP - I couldn't help but ask. What went wrong? Why can't we make it work?

The answer, it turns out, is surprisingly simple. The great design flaw was in how the famous "4-tuple" (source ip, source port, destination ip, destination port) was defined. We use the 4-tuple to identify a given TCP or UDP session; if a packet has those four fields the same, then it belongs to a given session, and we can deliver it to whatever socket is handling that session. But the 4-tuple crosses two layers: internetwork (layer 3) and transport (layer 4). If, instead, we had identified sessions using *only* layer 4 data, then mobile IP would have worked perfectly.

Let's do a quick example. X port 1111 is talking to Y port 80, so it sends a packet with 4-tuple (X,1111,Y,80). The response comes back with (Y,80,X,1111), and the kernel delivers it to the socket that generated the original packet. When X sends more packets tagged (X,1111,Y,80), then Y delivers them all to the same server socket, and so on.

Then, if X hops IP addresses, it gets a new name, say Q. Now it'll start sending packets with (Q,1111,Y,80). Y has no idea what that means, and throws it away. Meanwhile, if Y sends packets tagged (Y,80,X,1111), they get lost, because there is no longer an X to receive them.

Imagine now that we tagged sockets without reference to their IP address. For that to work, we'd need much bigger port numbers (which are currently 16 bits). Let's make them, say, 128 or 256 bits, some kind of unique hash.

Now X sends out packets to Y with tag (uuid,80). Note, the packets themselves still contain the (X,Y) addressing information, down at layer 3 - that's how they get routed to the right machine in the first place. But the kernel doesn't *use* the layer 3 information to decide which socket to deliver to; it just uses the uuid. The destination port (80 in this case) is only needed to initiate a new session, to identify what service you want to connect to, and can be ignored or left out after that.

For the return direction, Y's kernel caches the fact that packets for (uuid) go to IP address X, which is the address it most recently received (uuid) packets from.

Now imagine that X changes addresses to Q. It still sends out packets tagged with (uuid,80), to IP address Y, but now those packets come from address Q. On machine Y, it receives the packet and matches it to the socket associated with (uuid), notes that the packets for that socket are now coming from address Q, and updates its cache. Its return packets can now be sent, tagged as (uuid), back to Q instead of X. Everything works! (Modulo some care to prevent connection hijacking by impostors.[2])

There's only one catch: that's not how UDP and TCP work, and it's too late to update them. Updating UDP and TCP would be like updating IPv4 to IPv6; a project that sounded simple, back in the 1990s, but decades later, is less than half accomplished (and the first half was the easy part; the long tail is much harder).

The positive news is we may be able to hack around it with yet another layering violation. If we throw away TCP - it's getting rather old anyway - and instead use QUIC over UDP, then we can just stop using the UDP 4-tuple as a connection identifier at all. Instead, if the UDP port number is the "special mobility layer" port, we unwrap the content, which can be another packet with a proper uuid tag, match it to the right session, and deliver those packets to the right socket.

There's even more good news: the experimental QUIC protocol already, at least in theory, has the right packet structure to work like this. It turns out you need unique session identifiers (keys) anyhow if you want to use stateless packet encryption and authentication, which QUIC does. So, perhaps with not much work, QUIC could support transparent roaming. What a world that would be!

At that point, all we'd have to do is eliminate all remaining UDP and TCP from the Internet, and then we would definitely not need layer 2 bridging anymore, for real this time, and then we could get rid of broadcasts and MAC addresses and SDN and DHCP and all that stuff.

And then the Internet would be elegant again.

[1] **Edit 2017-08-16:** It turns out that nothing in this section requires IPv6. It would work fine with IPv4 and NAT, even roaming across multiple NATs.

[2] **Edit 2017-08-15:** Some people asked what "some care to prevent connection hijacking" might look like. There are various ways to do it, but the simplest would be to do something like the SYN-ACK-SYNACK exchange TCP does at connection startup. If Y just trusts the first packet from the new host Q, then it's too easy for any attacker to take over the X->Y connection by simply sending a packet to Y from anywhere on the Internet. (Although it's a bit hard to guess which 256-bit uuid to fill in.) But if Y sends back a cookie that Q must receive and process and send back to Y, that ensures that Q is at least a man-in-the-middle and not just an outside attacker (which is all TCP would guarantee anyway). If you're using an encrypted protocol (like QUIC[3]), the handshake can also be protected by your session key.

[3] **Edit 2017-10-24:** Besides QUIC, there are several other candidates for such a protocol, including MinimaLT. I didn't mention MinimaLT originally because it wasn't part of my original conversation with the IETF people, but I don't mean to imply that QUIC is the only possible option as a roaming-capable TCP replacement. In fact, MinimaLT is the first protocol I heard of that elegantly solved the roaming problem. Future solutions that might get adopted, including by QUIC, will likely be modeled after MinimaLT's solution.

Update 2020-07-09: I've posted more thoughts on IPv4/IPv6 migration and interoperability on the Tailscale blog.

Related
Content-Centric Networking (CCN) as an alternative to IP (2012)
IPv4, IPv6, and a sudden change in attitude (2020)
Unrelated
Indirection (2004)