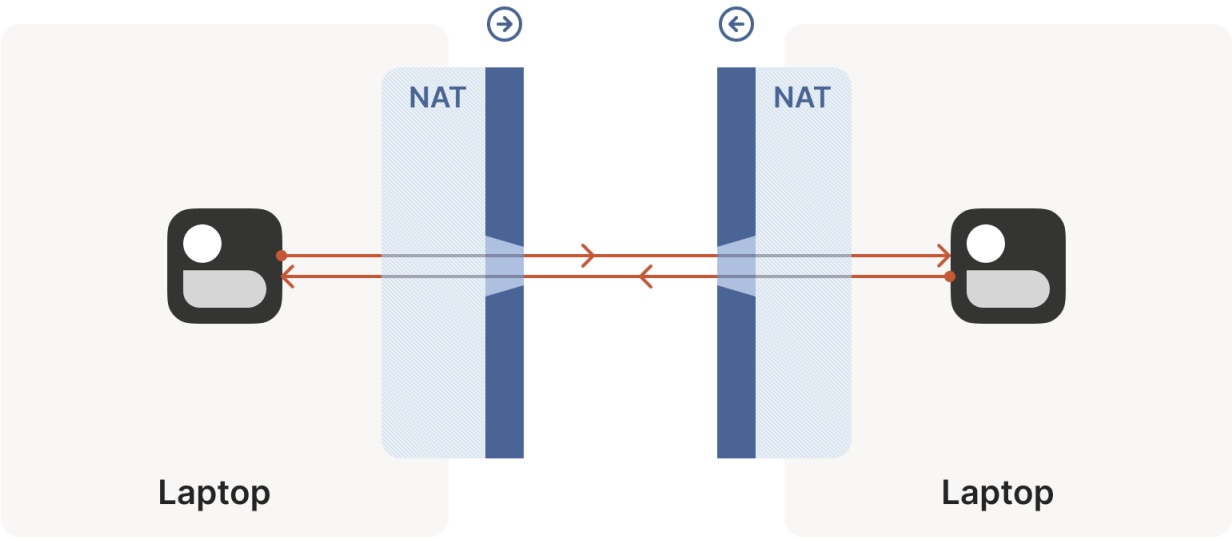
tailscale

Go back

# How NAT traversal works

August 21 2020 David Anderson

We covered a lot of ground in our post about *How Tailscale Works*. However, we glossed over how we can get through NATs (Network Address Translators) and connect your devices directly to each other, no matter what's standing between them. Let's talk about that now!




Let's start with a simple problem: establishing a peer-to-peer connection between two machines. In Tailscale's case, we want to set up a WireGuard® tunnel, but that doesn't really matter. The techniques we use are widely applicable and the work of many people over decades. For example, WebRTC uses this bag of tricks to send peer-to-peer audio, video and data between web browsers. VoIP phones and some video games use similar techniques, though not always successfully.

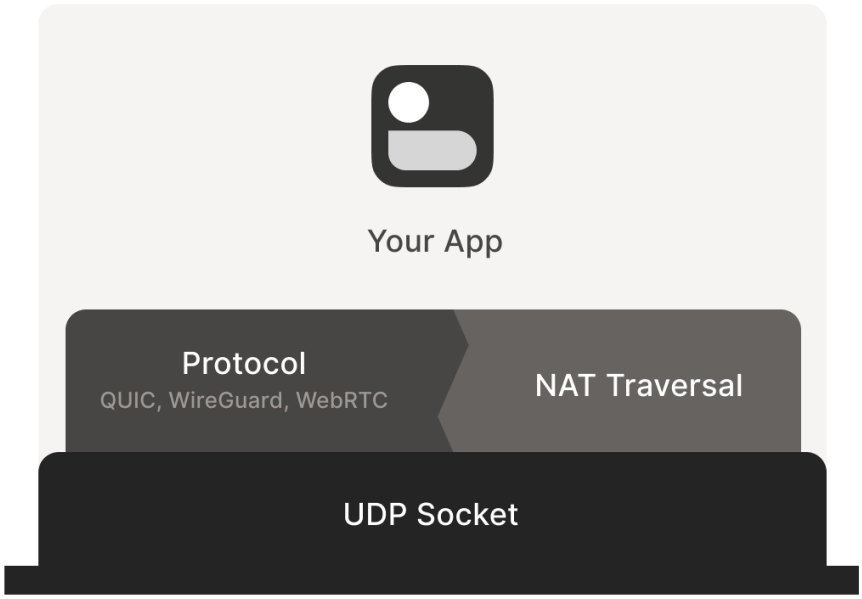
We'll be discussing these techniques generically, using Tailscale and others for examples where appropriate. Let's say you're making your own protocol and that you want NAT traversal. You need two things.

First, the protocol should be based on UDP. You *can* do NAT traversal with TCP, but it adds another layer of complexity to an already quite complex problem, and may even

tailscale

If you're reaching for TCP because you want a stream-oriented connection when the NAT traversal is done, consider using QUIC instead. It builds on top of UDP, so we can focus on UDP for NAT traversal and still have a nice stream protocol at the end.

Second, you need direct control over the network socket that's sending and receiving network packets. As a rule, you can't take an existing network library and make it traverse NATs, because you have to send and receive extra packets that aren't part of the "main" protocol you're trying to speak. Some protocols tightly integrate the NAT traversal with the rest (e.g. WebRTC). But if you're building your own, it's helpful to think of NAT traversal as a separate entity that shares a socket with your main protocol. Both run in parallel, one enabling the other.




Direct socket access may be tough depending on your situation. One workaround is to run a local proxy. Your protocol speaks to this proxy, and the proxy does both NAT traversal and relaying of your packets to the peer. This layer of indirection lets you benefit from NAT traversal without altering your original program.

With prerequisites out of the way, let's go through NAT traversal from first principles. Our goal is to get UDP packets flowing bidirectionally between two devices, so that our other protocol (WireGuard, QUIC, WebRTC, ...) can do something cool. There are two obstacles to having this Just Work: stateful firewalls and NAT devices.

## Figuring out firewalls

Stateful firewalls are the simpler of our two problems. In fact, most NAT devices include a stateful firewall, so we need to solve this subset before we can tackle NATs.

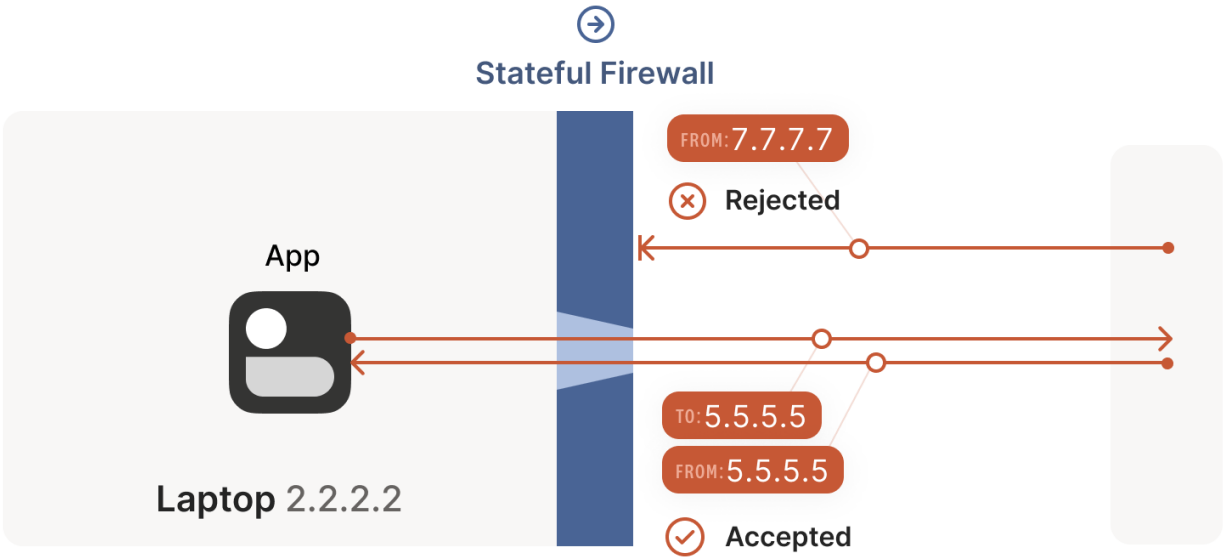
There are many incarnations to consider. Some you might recognize are the Windows Defender firewall, Ubuntu's ufw (using iptables/nftables), BSD's pf (also used by

tailscale

connections. There might be a few handpicked exceptions, such as allowing inbound SSH.

But connections and "direction" are a figment of the protocol designer's imagination. On the wire, every connection ends up being bidirectional; it's all individual packets flying back and forth. How does the firewall know what's inbound and what's outbound?

That's where the stateful part comes in. Stateful firewalls remember what packets they've seen in the past and can use that knowledge when deciding what to do with new packets that show up.



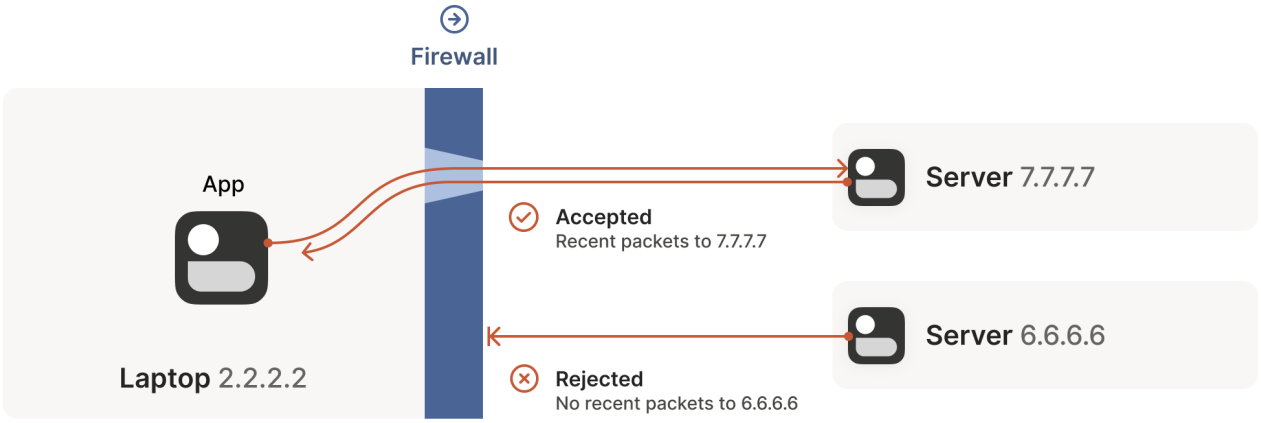

For UDP, the rule is very simple: the firewall allows an inbound UDP packet if it previously saw a matching outbound packet. For example, if our laptop firewall sees a UDP packet leaving the laptop from `2.2.2.2:1234` to `7.7.7.7:5678`, it'll make a note that incoming packets from `7.7.7.7:5678` to `2.2.2.2:1234` are also fine. The trusted side of the world clearly intended to communicate with `7.7.7.7:5678`, so we should let them talk back.

(As an aside, some *very* relaxed firewalls might allow traffic from anywhere back to `2.2.2.2:1234` once `2.2.2.2:1234` has communicated with anyone. Such firewalls make our traversal job easier, but are increasingly rare.)

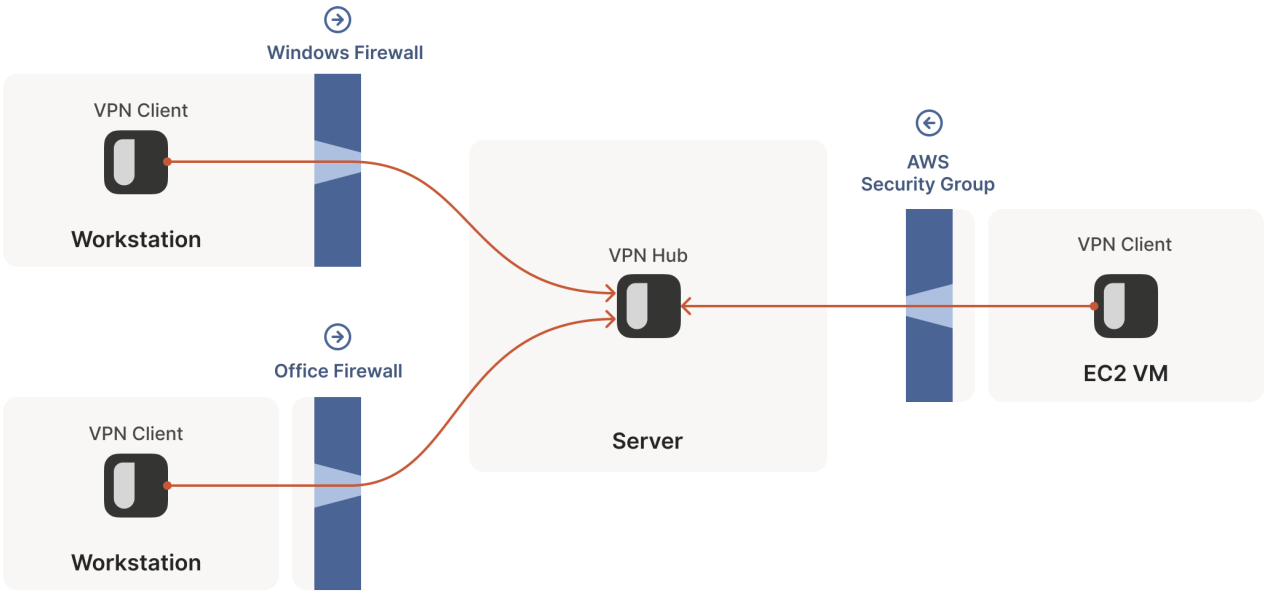
## Firewall face-off

This rule for UDP traffic is only a minor problem for us, as long as all the firewalls on the path are "facing" the same way. That's usually the case when you're communicating with a server on the internet. Our only constraint is that the machine

tailscale

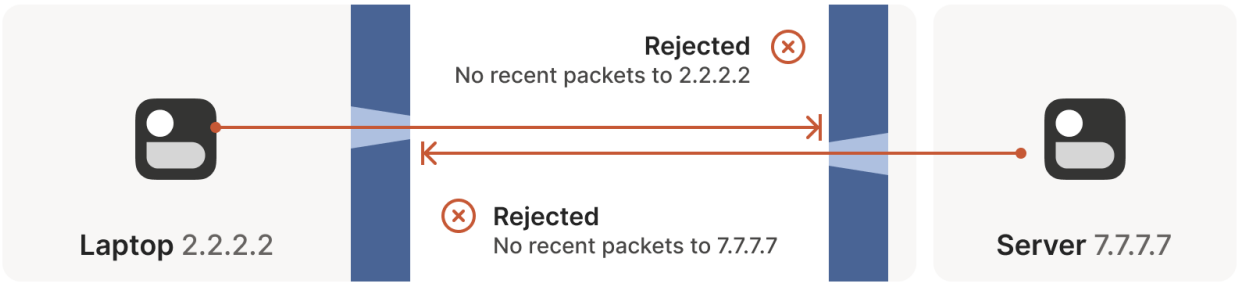



This is fine, but not very interesting: we've reinvented client/server communication, where the server makes itself easily reachable to clients. In the VPN world, this leads to a hub-and-spoke topology: the hub has no firewalls blocking access to it and the firewalled spokes connect to the hub.




The problems start when two of our "clients" want to talk directly. Now the firewalls are facing each other. According to the rule we established above, this means both sides must go first, but also that neither can go first, because the other side has to go first!

tailscale




How do we get around this? One way would be to require users to reconfigure one or both of the firewalls to "open a port" and allow the other machine's traffic. This is not very user friendly. It also doesn't scale to mesh networks like Tailscale, in which we expect the peers to be moving around the internet with some regularity. And, of course, in many cases you don't have control over the firewalls: you can't reconfigure the router in your favorite coffee shop, or at the airport. (At least, hopefully not!)

We need another option. One that doesn't involve reconfiguring firewalls.

## Finessing finicky firewalls

The trick is to carefully read the rule we established for our stateful firewalls. For UDP, the rule is: **packets must flow out before packets can flow back in.**

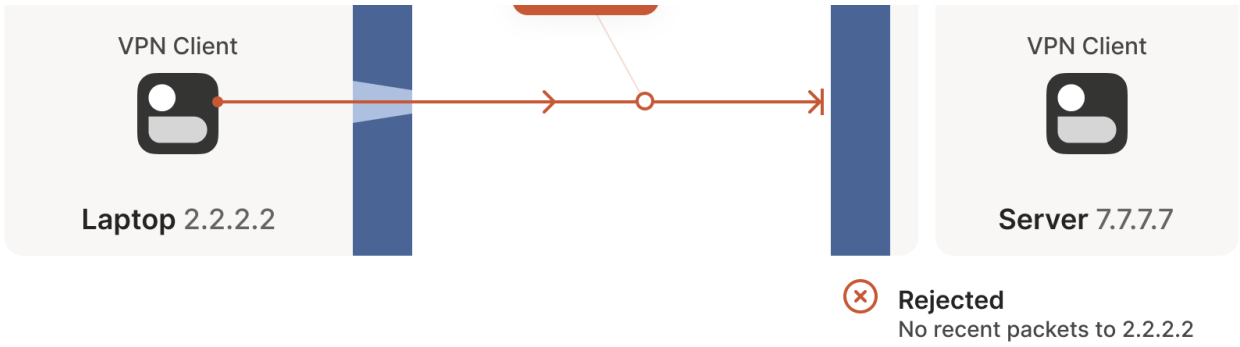
However, nothing says the packets must be *related* to each other beyond the IPs and ports lining up correctly. As long as *some* packet flowed outwards with the right source and destination, any packet that *looks like* a response will be allowed back in, even if the other side never received your packet!

So, to traverse these multiple stateful firewalls, we need to share some information to get underway: the peers have to know in advance the `ip:port` their counterpart is using. One approach is to statically configure each peer by hand, but this approach doesn't scale very far. To move beyond that, we built a coordination server to keep the `ip:port` information synchronized in a flexible, secure manner.

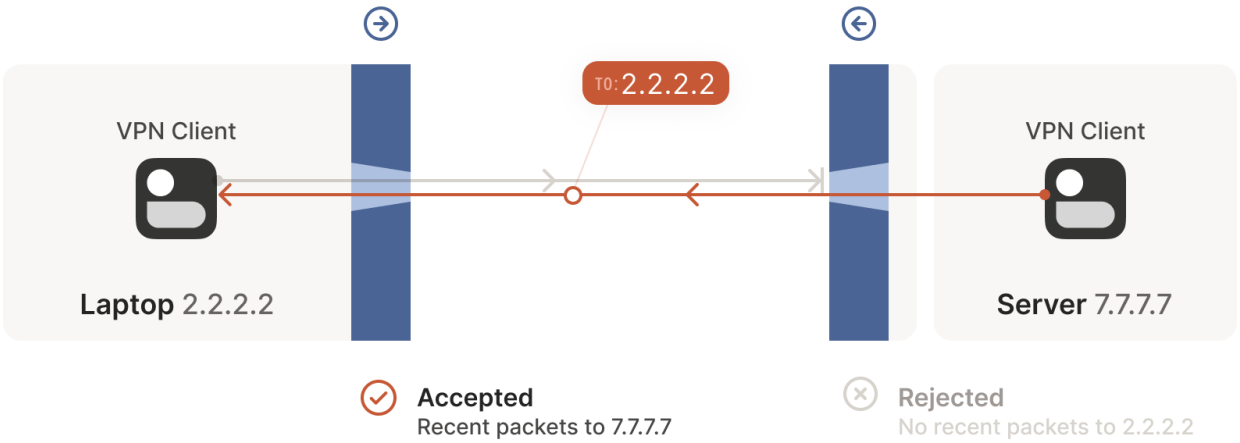
Then, the peers start sending UDP packets to each other. They must expect some of these packets to get lost, so they can't carry any precious information unless you're prepared to retransmit them. This is generally true of UDP, but especially true here. We're *going* to lose some packets in this process.

Our laptop and workstation are now listening on fixed ports, so that they both know exactly what `ip:port` to talk to. Let's take a look at what happens.

tailscale



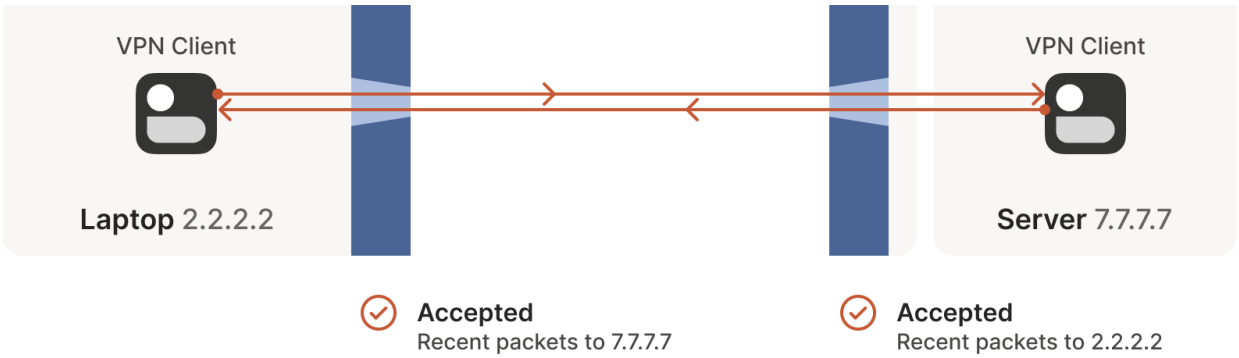

The laptop's first packet, from `2.2.2.2:1234` to `7.7.7.7:5678`, goes through the Windows Defender firewall and out to the internet. The corporate firewall on the other end blocks the packet, since it has no record of `7.7.7.7:5678` ever talking to `2.2.2.2:1234`. However, Windows Defender now remembers that it should expect and allow responses from `7.7.7.7:5678` to `2.2.2.2:1234`.




Next, the workstation's first packet from `7.7.7.7:5678` to `2.2.2.2:1234` goes through the corporate firewall and across the internet. When it arrives at the laptop, Windows Defender thinks "ah, a response to that outbound request I saw", and lets the packet through! Additionally, the corporate firewall now remembers that it should expect responses from `2.2.2.2:1234` to `7.7.7.7:5678`, and that those packets are also okay.

Encouraged by the receipt of a packet from the workstation, the laptop sends another packet back. It goes through the Windows Defender firewall, through the corporate firewall (because it's a "response" to a previously sent packet), and arrives at the workstation.

tailscale




Success! We've established two-way communication through a pair of firewalls that, at first glance, would have prevented it.

## Creative connectivity caveats

It's not always so easy. We're relying on some indirect influence over third-party systems, which requires careful handling. What do we need to keep in mind when managing firewall-traversing connections?

Both endpoints must attempt communication at roughly the same time, so that all the intermediate firewalls open up while both peers are still around. One approach is to have the peers retry continuously, but this is wasteful. Wouldn't it be better if both peers knew to start establishing a connection at the same time?

This may sound a little recursive: to communicate, first you need to be able to communicate. However, this preexisting "side channel" doesn't need to be very fancy: it can have a few seconds of latency, and only needs to deliver a few thousand bytes in total, so a tiny VM can easily be a matchmaker for thousands of machines.

In the distant past, I used XMPP chat messages as the side channel, with great results. As another example, WebRTC requires you to come up with your own "signalling channel" (a name that reveals WebRTC's IP telephony ancestry), and plug it into the WebRTC APIs. In Tailscale, our coordination server and fleet of DERP (Detour Encrypted Routing Protocol) servers act as our side channel.

Stateful firewalls have limited memory, meaning that we need periodic communication to keep connections alive. If no packets are seen for a while (a common value for UDP is 30 seconds), the firewall forgets about the session, and we have to start over. To avoid this, we use a timer and must either send packets regularly to reset the timers, or have some out-of-band way of restarting the connection on demand.

On the plus side, one thing we *don't* need to worry about is exactly how many firewalls exist between our two peers. As long as they are stateful and allow outbound connections, the simultaneous transmission technique will get through any number of

tailscale

...Right?

Well, not quite. For this to work, our peers need to know in advance what `ip:port` to use for their counterparts. This is where NATs come into play, and ruin our fun.

## The nature of NATs

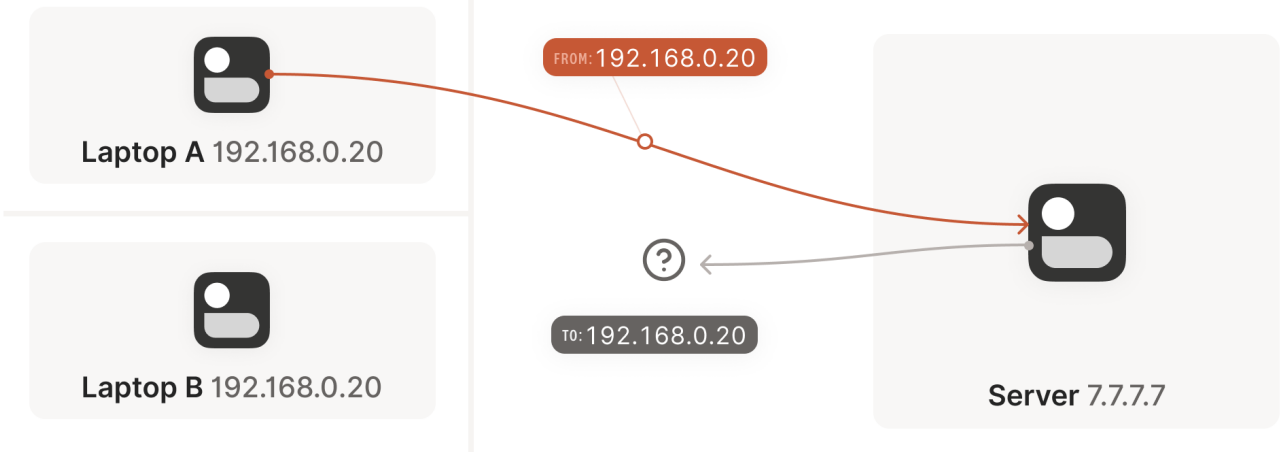
We can think of NAT (Network Address Translator) devices as stateful firewalls with one more really annoying feature: in addition to all the stateful firewalling stuff, they also alter packets as they go through.

A NAT device is anything that does any kind of Network Address Translation, i.e. altering the source or destination IP address or port. However, when talking about connectivity problems and NAT traversal, all the problems come from Source NAT, or SNAT for short. As you might expect, there is also DNAT (Destination NAT), and it's very useful but not relevant to NAT traversal.

The most common use of SNAT is to connect many devices to the internet, using fewer IP addresses than the number of devices. In the case of consumer-grade routers, we map all devices onto a single public-facing IP address. This is desirable because it turns out that there are way more devices in the world that want internet access, than IP addresses to give them (at least in IPv4 — we'll come to IPv6 in a little bit). NATs let us have many devices sharing a single IP address, so despite the global shortage of IPv4 addresses, we can scale the internet further with the addresses at hand.

### Navigating a NATty network

Let's look at what happens when your laptop is connected to your home Wi-Fi and talks to a server on the internet.



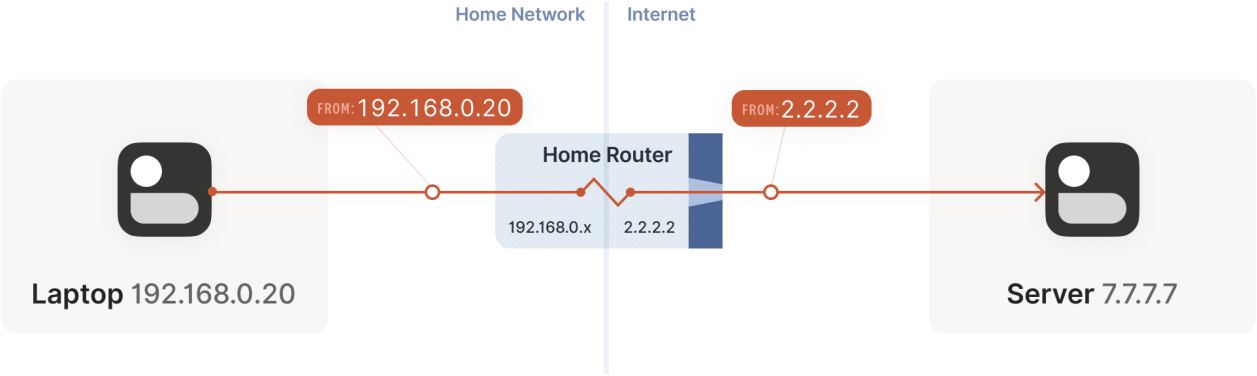
tailscale

`192.168.0.20` is a private IP address, which appears on many different peoples private networks. The internet won't know how to get responses back to us.

Enter the home router. The laptop's packets flow through the home router on their way to the internet, and the router sees that this is a new session that it's never seen before.

It knows that `192.168.0.20` won't fly on the internet, but it can work around that: it picks some unused UDP port on its own public IP address — we'll use `2.2.2.2:4242` — and creates a *NAT mapping* that establishes an equivalence: `192.168.0.20:1234` on the LAN side is the same as `2.2.2.2:4242` on the internet side.

From now on, whenever it sees packets that match that mapping, it will rewrite the IPs and ports in the packet appropriately.

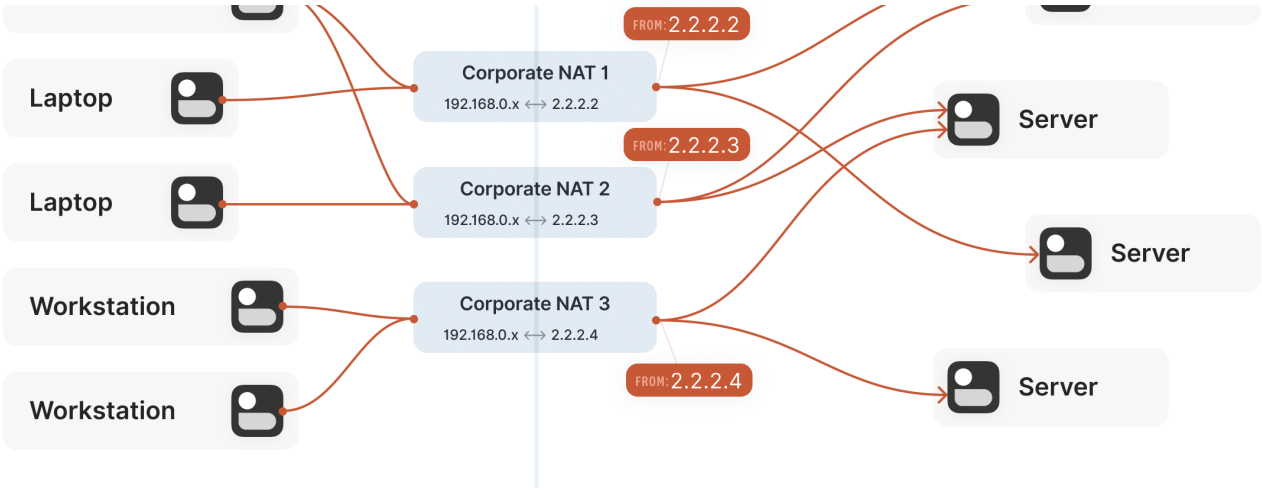



Resuming our packet's journey: the home router applies the NAT mapping it just created, and sends the packet onwards to the internet. Only now, the packet is from `2.2.2.2:4242`, not `192.168.0.20:1234`. It goes on to the server, which is none the wiser. It's communicating with `2.2.2.2:4242`, like in our previous examples sans NAT.

Responses from the server flow back the other way as you'd expect, with the home router rewriting `2.2.2.2:4242` back to `192.168.0.20:1234`. The laptop is *also* none the wiser, from its perspective the internet magically figured out what to do with its private IP address.

Our example here was with a home router, but the same principle applies on corporate networks. The usual difference there is that the NAT layer consists of multiple machines (for high availability or capacity reasons), and they can have more than one public IP address, so that they have more public `ip:port` combinations to choose from and can sustain more active clients at once.

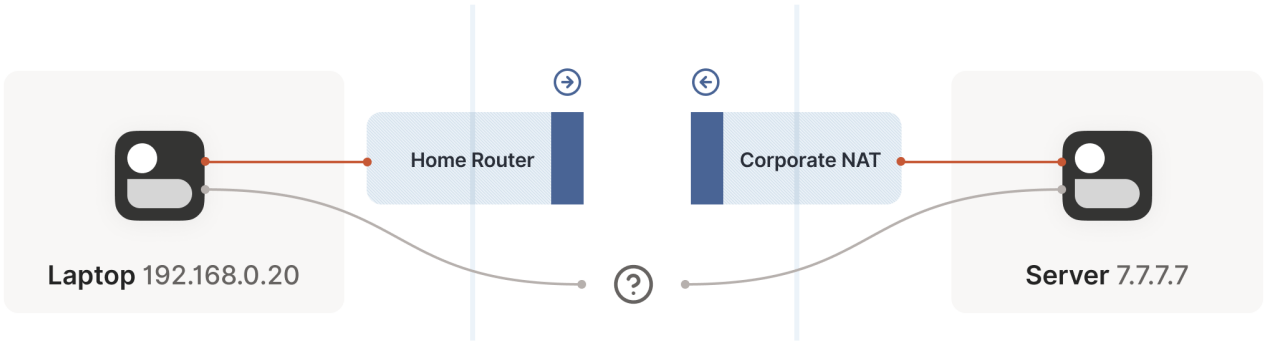
tailscale




Multiple NATs on a single layer allow for higher availability or capacity, but function the same as a single NAT.

## A study in STUN

We now have a problem that looks like our earlier scenario with the stateful firewalls, but with NAT devices:




Our problem is that our two peers don't know what the `ip:port` of their peer is. Worse, strictly speaking there is *no* `ip:port` until the other peer sends packets, since NAT mappings only get created when outbound traffic towards the internet requires it. We're back to our stateful firewall problem, only worse: both sides have to speak first, but neither side knows to whom to speak, and can't know until the other side speaks first.

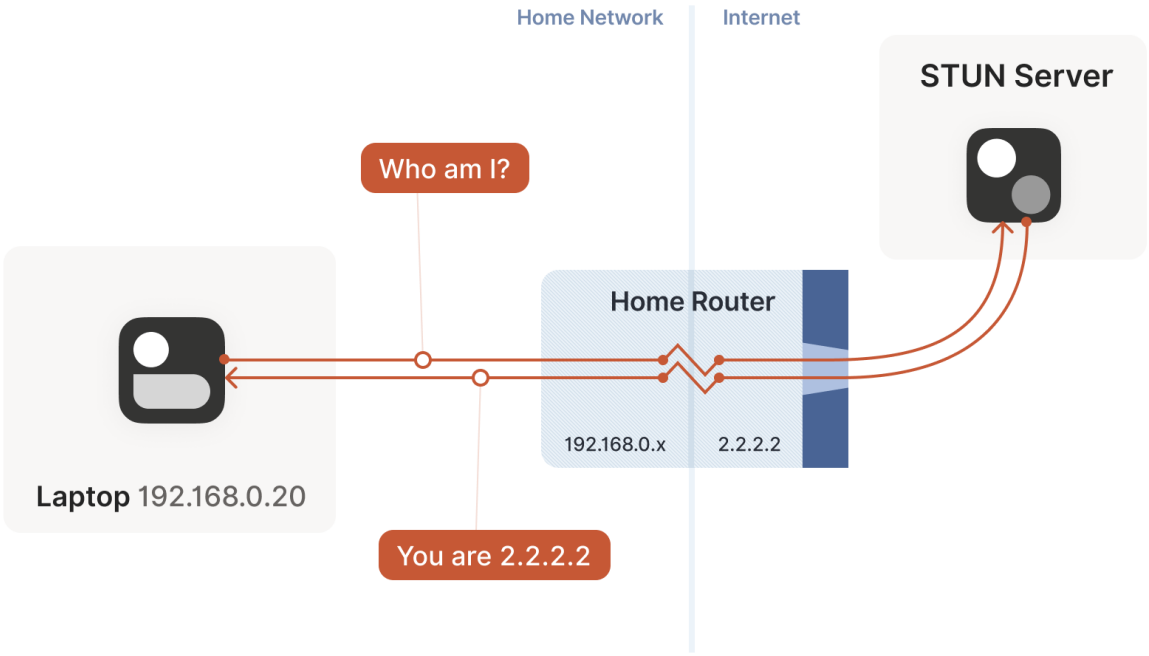
How do we break the deadlock? That's where STUN comes in. STUN is both a set of studies of the detailed behavior of NAT devices, and a protocol that aids in NAT traversal. The main thing we care about for now is the network protocol.

STUN relies on a simple observation: when you talk to a server on the internet from a NATed client, the server sees the public `ip:port` that your NAT device created for you, not your LAN `ip:port`. So, the server can *tell* you what `ip:port` it saw. That way,

tailscale

back to our "simple" case of firewall traversal.

That's fundamentally all that the STUN protocol is: your machine sends a "what's my endpoint from your point of view?" request to a STUN server, and the server replies with "here's the `ip:port` that I saw your UDP packet coming from."




(The STUN protocol has a bunch more stuff in it — there's a way of obfuscating the `ip:port` in the response to stop really broken NATs from mangling the packet's payload, and a whole authentication mechanism that only really gets used by TURN and ICE, sibling protocols to STUN that we'll talk about in a bit. We can ignore all of that stuff for address discovery.)

Incidentally, this is why we said in the introduction that, if you want to implement this yourself, the NAT traversal logic and your main protocol have to share a network socket. Each socket gets a different mapping on the NAT device, so in order to discover your public `ip:port`, you have to send and receive STUN packets from the socket that you intend to use for communication, otherwise you'll get a useless answer.

## How this helps

Given STUN as a tool, it seems like we're close to done. Each machine can do STUN to discover the public-facing `ip:port` for its local socket, tell its peers what that is, everyone does the firewall traversal stuff, and we're all set... Right?

Well, it's a mixed bag. This'll work in some cases, but not others. Generally speaking, this'll work with most home routers, and will fail with some corporate NAT gateways.

tailscale

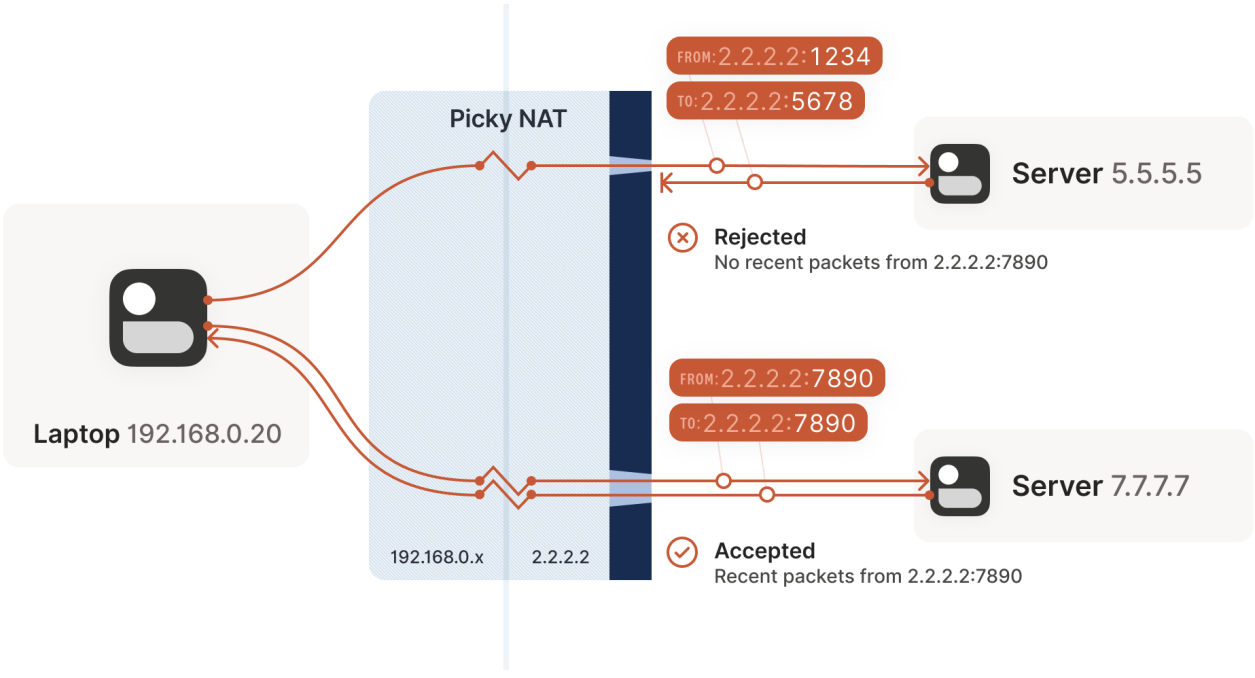
a rant for another time.)

The problem is an assumption we made earlier: when the STUN server told us that we're `2.2.2.2:4242` from its perspective, we assumed that meant that we're `2.2.2.2:4242` from the entire internet's perspective, and that therefore anyone can reach us by talking to `2.2.2.2:4242`.

As it turns out, that's not always true. Some NAT devices behave exactly in line with our assumptions. Their stateful firewall component still wants to see packets flowing in the right order, but we can reliably figure out the correct `ip:port` to give to our peer and do our simultaneous transmission trick to get through. Those NATs are great, and our combination of STUN and the simultaneous packet sending will work fine with those.

(in theory, there are also NAT devices that are super relaxed, and don't ship with stateful firewall stuff at all. In those, you don't even need simultaneous transmission, the STUN request gives you an internet `ip:port` that anyone can connect to with no further ceremony. If such devices do still exist, they're increasingly rare.)

Other NAT devices are more difficult, and create a completely different NAT mapping for every different destination that you talk to. On such a device, if we use the same socket to send to `5.5.5.5:1234` and `7.7.7.7:2345`, we'll end up with two different ports on 2.2.2.2, one for each destination. If you use the wrong port to talk back, you don't get through.




## Naming our NATs

tailscale

might have heard of "Full Cone", "Restricted Cone", "Port-Restricted Cone" and "Symmetric" NATs. These are terms that come from early research into NAT traversal.

That terminology is honestly quite confusing. I always look up what a Restricted Cone NAT is supposed to be. Empirically, I'm not alone in this, because most of the internet calls "easy" NATs Full Cone, when these days they're much more likely to be Port-Restricted Cone.

More recent research and RFCs have come up with a much better taxonomy. First of all, they recognize that there are many more varying dimensions of behavior than the single "cone" dimension of earlier research, so focusing on the cone-ness of your NAT isn't necessarily helpful. Second, they came up with words that more plainly convey what the NAT is doing.

The "easy" and "hard" NATs above differ in a single dimension: whether or not their NAT mappings depend on what the destination is. RFC 4787 calls the easy variant "Endpoint-Independent Mapping" (EIM for short), and the hard variant "Endpoint-Dependent Mapping" (EDM for short). There's a subcategory of EDM that specifies whether the mapping varies only on the destination IP, or on both the destination IP and port. For NAT traversal, the distinction doesn't matter. Both kinds of EDM NATs are equally bad news for us.

In the grand tradition of naming things being hard, endpoint-independent NATs still depend on an endpoint: each *source* `ip:port` gets a different mapping, because otherwise your packets would get mixed up with someone else's packets, and that would be chaos. Strictly speaking, we should say "Destination Endpoint Independent Mapping" (DEIM?), but that's a mouthful, and since "Source Endpoint Independent Mapping" would be another way to say "broken", we don't specify. Endpoint always means "Destination Endpoint."

You might be wondering how 2 kinds of endpoint dependence maps into 4 kinds of cone-ness. The answer is that cone-ness encompasses two orthogonal dimensions of NAT behavior. One is NAT mapping behavior, which we looked at above, and the other is stateful firewall behavior. Like NAT mapping behavior, the firewalls can be Endpoint-Independent or a couple of variants of Endpoint-Dependent. If you throw all of these into a matrix, you can reconstruct the cone-ness of a NAT from its more fundamental properties:

NAT Cone Types

| | Endpoint-Independent NAT mapping | Endpoint-Dependent NAT mapping (all types) |
|---|---|---|
| **Endpoint-Independent firewall** | Full Cone NAT | N/A* |

tailscale

| | | |
|---|---|---|
| **Endpoint-Dependent firewall (dest. IP only)** | Restricted Cone NAT | N/A* |
| **Endpoint-Dependent firewall (dest. IP+port)** | Port-Restricted Cone NAT | Symmetric NAT |

* can theoretically exist, but don't show up in the wild

Once broken down like this, we can see that cone-ness isn't terribly useful to us. The major distinction we care about is Symmetric versus anything else — in other words, we care about whether a NAT device is EIM or EDM.

While it's neat to know exactly how your firewall behaves, we don't care from the point of view of writing NAT traversal code. Our simultaneous transmission trick will get through all three variants of firewalls. In the wild we're overwhelmingly dealing only with IP-and-port endpoint-dependent firewalls. So, for practical code, we can simplify the table down to:

| | Endpoint-Independent NAT mapping | Endpoint-Dependent NAT mapping (dest. IP only) |
|---|---|---|
| **Firewall is yes** | Easy NAT | Hard NAT |

If you'd like to read more about the newer taxonomies of NATs, you can get the full details in RFCs 4787 (NAT Behavioral Requirements for UDP), 5382 (for TCP) and 5508 (for ICMP). And if you're implementing a NAT device, these RFCs are also your guide to what behaviors you *should* implement, to make them well-behaved devices that play well with others and don't generate complaints about Halo multiplayer not working.

Back to our NAT traversal. We were doing well with STUN and firewall traversal, but these hard NATs are a big problem. It only takes one of them in the whole path to break our current traversal plans.

But wait, this post is titled "how NAT traversal works", not "how NAT traversal doesn't work." So presumably, I have a trick up my sleeve to get out of this, right?

## Have you considered giving up?

This is a good time to have the awkward part of our chat: what happens when we empty our entire bag of tricks, and we *still* can't get through? A lot of NAT traversal code out there gives up and declares connectivity impossible. That's obviously not acceptable for us; Tailscale is nothing without the connectivity.

tailscale

Sort of. It's certainly not as good as a direct connection, but if the relay is "near enough" to the network path your direct connection would have taken, and has enough bandwidth, the impact on your connection quality isn't huge. There will be a bit more latency, maybe less bandwidth. That's still much better than no connection at all, which is where we were heading.

And keep in mind that we only resort to this in cases where direct connections fail. We can still establish direct connections through a *lot* of different networks. Having relays to handle the long tail isn't that bad.

Additionally, some networks can break our connectivity much more directly than by having a difficult NAT. For example, we've observed that the UC Berkeley guest Wi-Fi blocks all outbound UDP except for DNS traffic. No amount of clever NAT tricks is going to get around the firewall eating your packets. So, we need some kind of reliable fallback no matter what.

You could implement relays in a variety of ways. The classic way is a protocol called TURN (Traversal Using Relays around NAT). We'll skip the protocol details, but the idea is that you authenticate yourself to a TURN server on the internet, and it tells you "okay, I've allocated `ip:port`, and will relay packets for you." You tell your peer the TURN `ip:port`, and we're back to a completely trivial client/server communication scenario.

For Tailscale, we didn't use TURN for our relays. It's not a particularly pleasant protocol to work with, and unlike STUN there's no real interoperability benefit since there are no open TURN servers on the internet.

Instead, we created DERP (Detoured Encrypted Routing Protocol), which is a general purpose packet relaying protocol. It runs over HTTP, which is handy on networks with strict outbound rules, and relays encrypted payloads based on the destination's public key.

As we briefly touched on earlier, we use this communication path both as a data relay when NAT traversal fails (in the same role as TURN in other systems) and as the side channel to help with NAT traversal. DERP is both our fallback of last resort to get connectivity, and our helper to upgrade to a peer-to-peer connection, when that's possible.

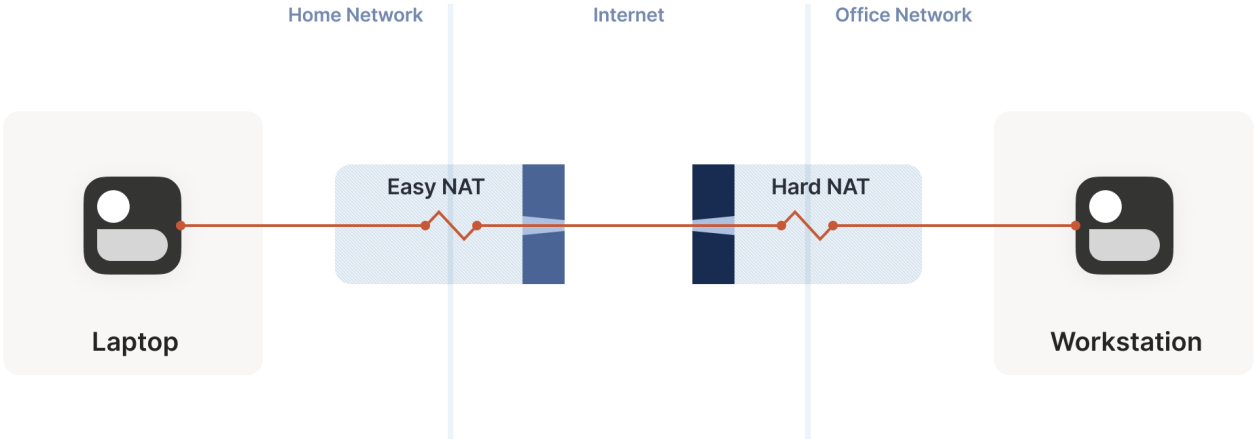
Now that we have a relay, in addition to the traversal tricks we've discussed so far, we're in pretty good shape. We can't get through *everything* but we can get through quite a lot, and we have a backup for when we fail. If you stopped reading now and implemented just the above, I'd estimate you could get a direct connection over 90% of the time, and your relays guarantee *some* connectivity all the time.

tailscale

What follows is a somewhat miscellaneous set of tricks, which can help us out in specific situations. None of them will solve NAT traversal by itself, but by combining them judiciously, we can get incrementally closer to a 100% success rate.

## The benefits of birthdays

Let's revisit our problem with hard NATs. The key issue is that the side with the easy NAT doesn't know what `ip:port` to send to on the hard side. But *must* send to the right `ip:port` in order to open up its firewall to return traffic. What can we do about that?




Well, we know *some* `ip:port` for the hard side, because we ran STUN. Let's assume that the IP address we got is correct. That's not *necessarily* true, but let's run with the assumption for now. As it turns out, it's mostly safe to assume this. (If you're curious why, see REQ-2 in RFC 4787.)

If the IP address is correct, our only unknown is the port. There's 65,535 possibilities... Could we try all of them? At 100 packets/sec, that's a worst case of 10 minutes to find the right one. It's better than nothing, but not great. And it *really* looks like a port scan (because in fairness, it is), which may anger network intrusion detection software.

We can do much better than that, with the help of the birthday paradox. Rather than open 1 port on the hard side and have the easy side try 65,535 possibilities, let's open, say, 256 ports on the hard side (by having 256 sockets sending to the easy side's `ip:port`), and have the easy side probe target ports at random.

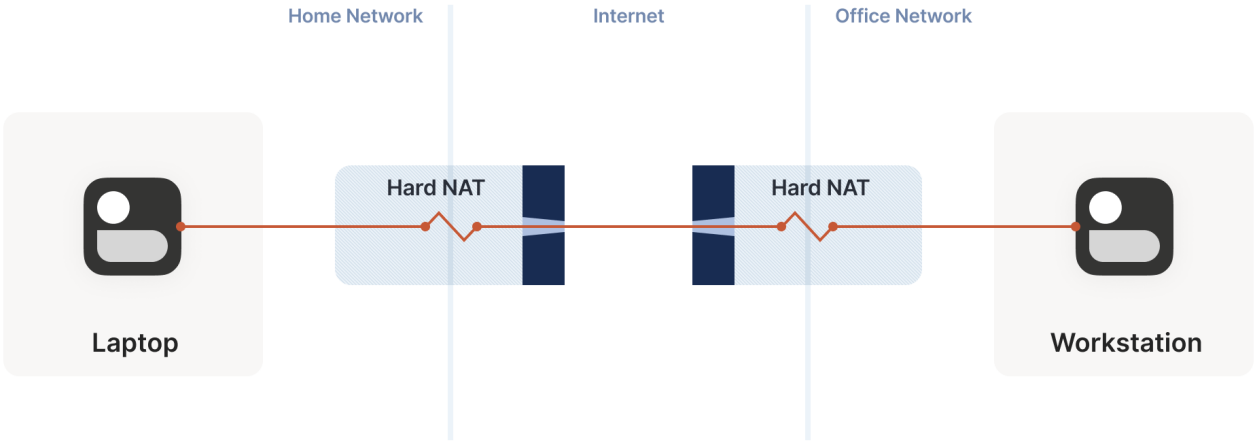
I'll spare you the detailed math, but you can check out the dinky python calculator I made while working it out. The calculation is a very slight variant on the "classic" birthday paradox, because it's looking at collisions between two sets containing distinct elements, rather than collisions within a single set. Fortunately, the difference works out slightly in our favor! Here's the chances of a collision of open ports (i.e.

tailscale

| Number of random probes | Chance of success |
|---|---|
| 174 | 50% |
| 256 | 64% |
| 1024 | 98% |
| 2048 | 99.9% |

If we stick with a fairly modest probing rate of 100 ports/sec, half the time we'll get through in under 2 seconds. And even if we get unlucky, 20 seconds in we're virtually guaranteed to have found a way in, after probing less than 4% of the total search space.

That's great! With this additional trick, one hard NAT in the path is an annoying speedbump, but we can manage. What about two?




We can try to apply the same trick, but now the search is much harder: each random destination port we probe through a hard NAT also results in a random *source* port. That means we're now looking for a collision on a `{source port, destination port}` pair, rather than just the destination port.

Again I'll spare you the calculations, but after 20 seconds in the same regime as the previous setup (256 probes from one side, 2048 from the other), our chance of success is... 0.01%.

This shouldn't be surprising if you've studied the birthday paradox before. The birthday paradox lets us convert `N` "effort" into something on the order of `sqrt(N)`. But we squared the size of the search space, so even the reduced amount of effort is still a lot more effort. To hit a 99.9% chance of success, we need each side to send 170,000 probes. At 100 packets/sec, that's 28 minutes of trying before we can

tailscale

you'd it would take without the birthday paradox.

In some applications, 28 minutes might still be worth it. Spend half an hour brute-forcing your way through, then you can keep pinging to keep the open path alive indefinitely — or at least until one of the NATs reboots and dumps all its state, then you're back to brute forcing. But it's not looking good for any kind of interactive connectivity.

Worse, if you look at common office routers, you'll find that they have a surprisingly low limit on active sessions. For example, a Juniper SRX 300 maxes out at 64,000 active sessions. We'd consume its entire session table with our one attempt to get through! And that's assuming the router behaves gracefully when overloaded. *And* this is all to get a single connection! What if we have 20 machines doing this behind the same router? Disaster.

Still, with this trick we can make it through a slightly harder network topology than before. That's a big deal, because home routers tend to be easy NATs, and hard NATs tend to be office routers or cloud NAT gateways. That means this trick buys us improved connectivity for the home-to-office and home-to-cloud scenarios, as well as a few office-to-cloud and cloud-to-cloud scenarios.

## Partially manipulating port maps

Our hard NATs would be so much easier if we could ask the NATs to stop being such jerks, and let more stuff through. Turns out, there's a protocol for that! Three of them, to be precise. Let's talk about port mapping protocols.

The oldest is the UPnP IGD (Universal Plug'n'Play Internet Gateway Device) protocol. It was born in the late 1990's, and as such uses a lot of very 90's technology (XML, SOAP, multicast HTTP over UDP — yes, really) and is quite hard to implement correctly and securely — but a lot of routers shipped with UPnP, and a lot still do. If we strip away all the fluff, we find a very simple request-response that all three of our port mapping protocols implement: "Hi, please forward a WAN port to `lan-ip:port`," and "okay, I've allocated `wan-ip:port` for you."

Speaking of stripping away the fluff: some years after UPnP IGD came out, Apple launched a competing protocol called NAT-PMP (NAT Port Mapping Protocol). Unlike UPnP, it *only* does port forwarding, and is extremely simple to implement, both on clients and on NAT devices. A little bit after that, NAT-PMP v2 was reborn as PCP (Port Control Protocol).

So, to help our connectivity further, we can look for UPnP IGD, NAT-PMP and PCP on our local default gateway. If one of the protocols elicits a response, we request a public port mapping. You can think of this as a sort of supercharged STUN: in addition to discovering our public `ip:port`, we can instruct the NAT to be friendlier to our

tailscale

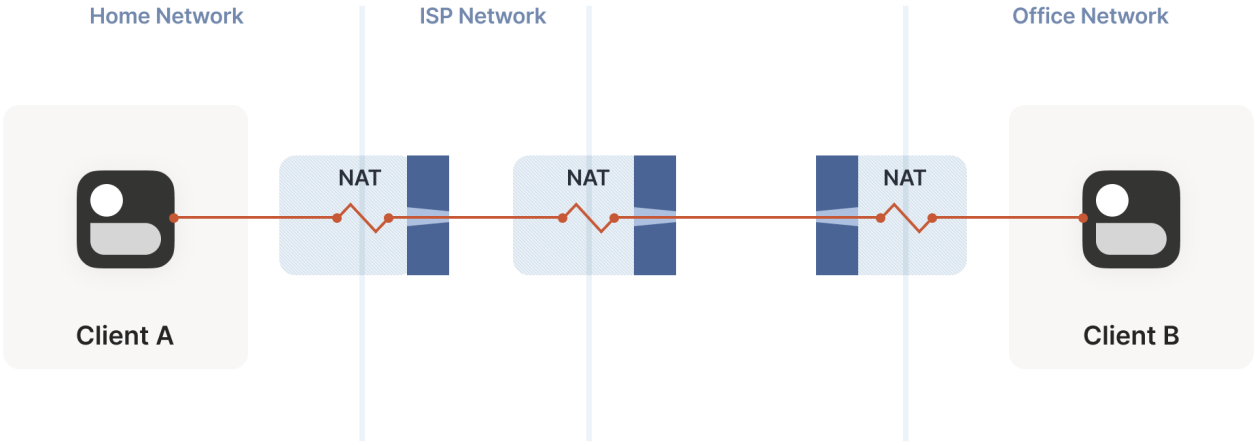
You can't rely on these protocols being present. They might not be implemented on your devices. They might be disabled by default and nobody knew to turn them on. They might have been disabled by policy.

Disabling by policy is fairly common because UPnP suffered from a number of high-profile vulnerabilities (since fixed, so newer devices can safely offer UPnP, if implemented properly). Unfortunately, many devices come with a single "UPnP" checkbox that actually toggles UPnP, NAT-PMP and PCP all at once, so folks concerned about UPnP's security end up disabling the perfectly safe alternatives as well.

Still, when it's available, it effectively makes one NAT vanish from the data path, which usually makes connectivity trivial... But let's look at the unusual cases.

## Negotiating numerous NATs

So far, the topologies we've looked at have each client behind one NAT device, with the two NATs facing each other. What happens if we build a "double NAT", by chaining two NATs in front of one of our machines?




What happens if we build a "double NAT", by chaining two NATs in front of one of our machines?

In this example, not much of interest happens. Packets from client A go through two different layers of NAT on their way to the internet. But the outcome is the same as it was with multiple layers of stateful firewalls: the extra layer is invisible to everyone, and our other techniques will work fine regardless of how many layers there are. All that matters is the behavior of the "last" layer before the internet, because that's the one that our peer has to find a way through.

The big thing that breaks is our port mapping protocols. They act upon the layer of NAT closest to the client, whereas the one we need to influence is the one furthest

tailscale

protocols give you enough information to find the "next NAT up" to repeat the process there, although you could try your luck with a traceroute and some blind requests to the next few hops.

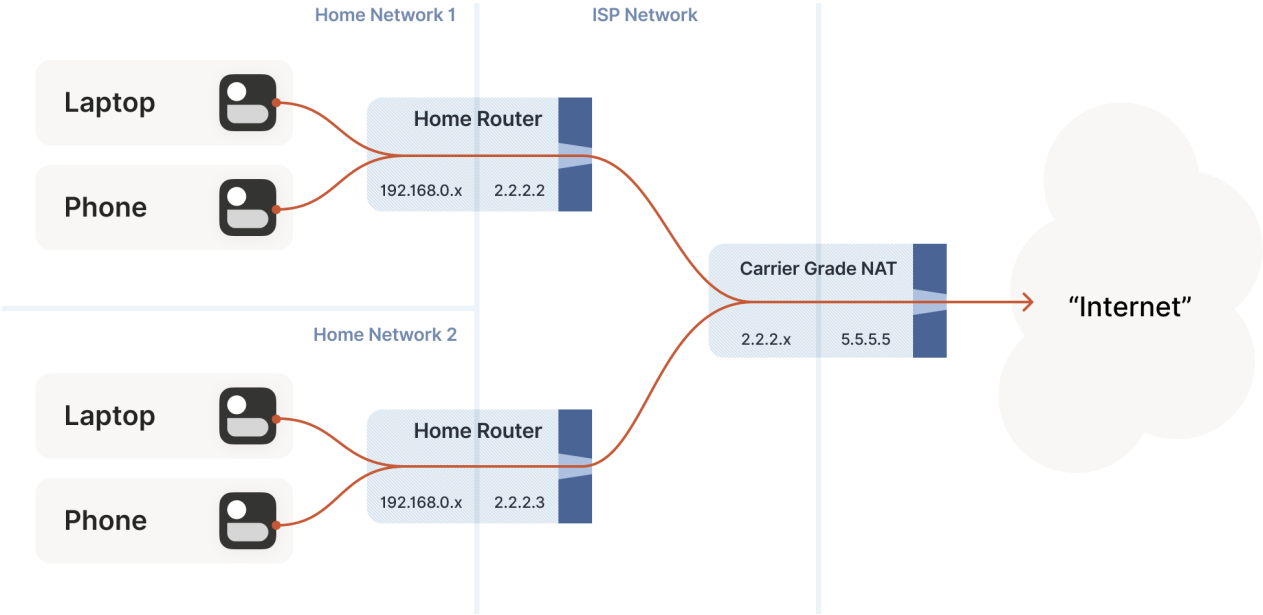
Breaking port mapping protocols is the reason why the internet is so full of warnings about the evils of double-NAT, and how you should bend over backwards to avoid them. But in fact, double-NAT is entirely invisible to most internet-using applications, because most applications don't try to do this kind of explicit NAT traversal.

I'm definitely not saying that you *should* set up a double-NAT in your network. Breaking the port mapping protocols will degrade multiplayer on many video games, and will likely strip IPv6 from your network, which robs you of some very good options for NAT-free connectivity. But, if circumstances beyond your control force you into a double-NAT, and you can live with the downsides, most things will still work fine.

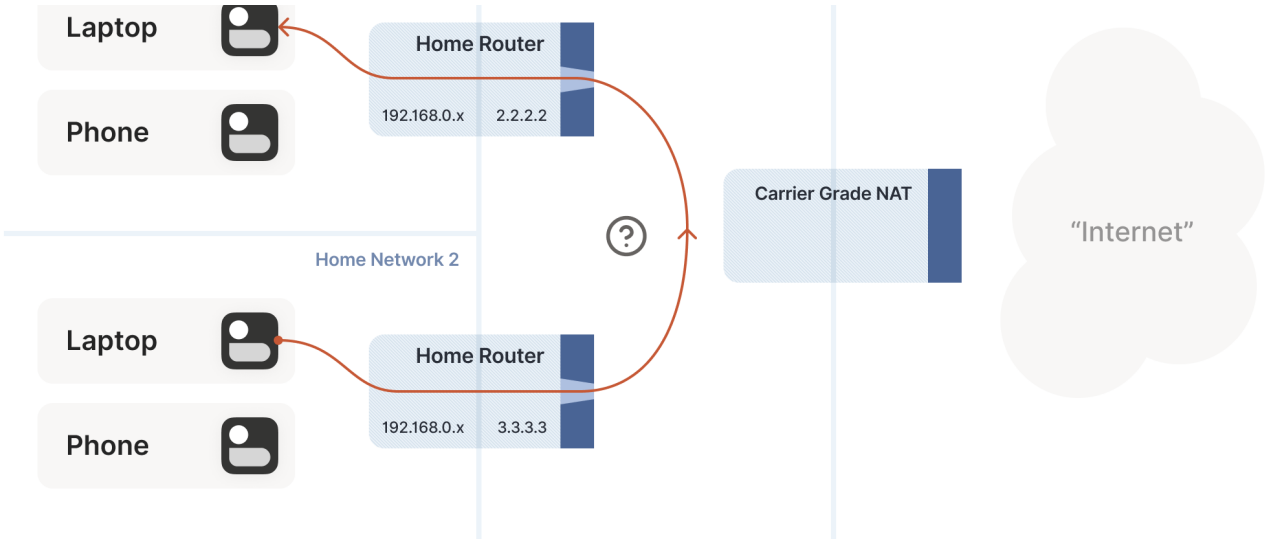
Which is a good thing, because you know what circumstances beyond your control force you to double-NAT? Let's talk carrier-grade NAT.

## Concerning CGNATs

Even with NATs to stretch the supply of IPv4 addresses, we're *still* running out, and ISPs can no longer afford to give one entire public IP address to every home on their network. To work around this, ISPs apply SNAT recursively: your home router SNATs your devices to an "intermediate" IP address, and further out in the ISP's network a second layer of NAT devices map those intermediate IPs onto a smaller number of public IPs. This is "carrier-grade NAT", or CGNAT for short.



tailscale




How do we connect two peers who are behind the same CGNAT, but different home NATs within?

Carrier-grade NAT is an important development for NAT traversal. Prior to CGNAT, enterprising users could work around NAT traversal difficulties by manually configuring port forwarding on their home routers. But you can't reconfigure the ISP's CGNAT! Now even power users have to wrestle with the problems NATs pose.

The good news: this is a run of the mill double-NAT, and so as we covered above it's mostly okay. Some stuff won't work as well as it could, but things work well enough that ISPs can charge money for it. Aside from the port mapping protocols, everything from our current bag of tricks works fine in a CGNAT world.

We do have to overcome a new challenge, however: how do we connect two peers who are behind the same CGNAT, but different home NATs within? That's how we set up peers A and B in the diagram above.

The problem here is that STUN doesn't work the way we'd like. We'd like to find out our `ip:port` on the "middle network", because it's effectively playing the role of a miniature internet to our two peers. But STUN tells us what our `ip:port` is from the STUN server's point of view, and the STUN server is out on the internet, beyond the CGNAT.

If you're thinking that port mapping protocols can help us here, you're right! If either peer's home NAT supports one of the port mapping protocols, we're happy, because we have an `ip:port` that behaves like an un-NATed server, and connecting is trivial. Ironically, the fact that double NAT "breaks" the port mapping protocols helps us! Of course, we still can't count on these protocols helping us out, doubly so because CGNAT ISPs tend to turn them off in the equipment they put in homes in order to avoid software getting confused by the "wrong" results they would get.

tailscale

CGNAT, so let's say that STUN tells us that peer A is `2.2.2.2:1234`, and peer B is `2.2.2.2:5678`.

The question is: what happens when peer A sends a packet to `2.2.2.2:5678`? We might hope that the following takes place in the CGNAT box:

Apply peer A's NAT mapping, rewrite the packet to be from `2.2.2.2:1234` and to `2.2.2.2:5678`.

Notice that `2.2.2.2:5678` matches peer B's *incoming* NAT mapping, rewrite the packet to be from `2.2.2.2:1234` and to peer B's private IP.

Send the packet on to peer B, on the "internal" interface rather than off towards the internet.

This behavior of NATs is called hairpinning, and with all this dramatic buildup you won't be surprised to learn that hairpinning works on some NATs and not others.

In fact, a great many otherwise well-behaved NAT devices don't support hairpinning, because they make assumptions like "a packet from my internal network to a non-internal IP address will always flow outwards to the internet", and so end up dropping packets as they try to turn around within the router. These assumptions might even be baked into routing silicon, where it's impossible to fix without new hardware.

Hairpinning, or lack thereof, is a trait of all NATs, not just CGNATs. In most cases, it doesn't matter, because you'd expect two LAN devices to talk directly to each other rather than hairpin through their default gateway. And it's a pity that it usually doesn't matter, because that's probably why hairpinning is commonly broken.

But once CGNAT is involved, hairpinning becomes vital to connectivity. Hairpinning lets you apply the same tricks that you use for internet connectivity, without worrying about whether you're behind a CGNAT. If both hairpinning and port mapping protocols fail, you're stuck with relaying.

## Ideally IPv6, NAT64 notwithstanding

By this point I expect some of you are shouting at your screens that the solution to all this nonsense is IPv6. All this is happening because we're running out of IPv4 addresses, and we keep piling on NATs to work around that. A much simpler fix would be to not have an IP address shortage, and make every device in the world reachable without NATs. Which is exactly what IPv6 gets us.

And you're right! Sort of. It's true that in an IPv6-only world, all of this becomes much simpler. Not trivial, mind you, because we're still stuck with stateful firewalls. Your office workstation may have a globally reachable IPv6 address, but I'll bet there's still

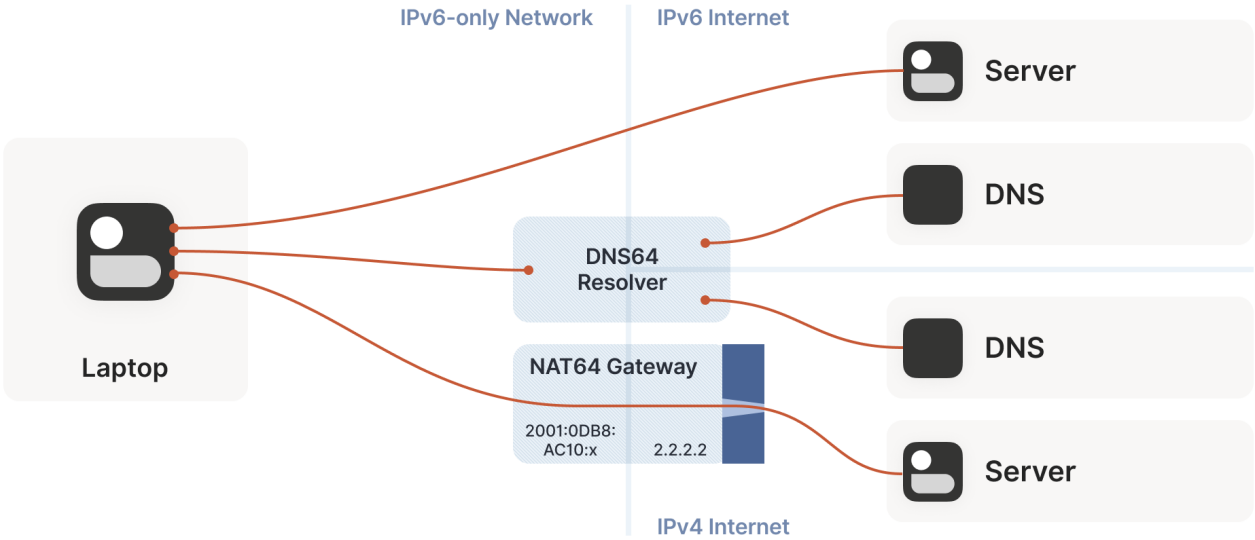
tailscale

So, we still need the firewall traversal stuff from the start of the article, and a side channel so that peers can know what `ip:port` to talk to. We'll probably also still want fallback relays that use a well-like protocol like HTTP, to get out of networks that block outbound UDP. But we can get rid of STUN, the birthday paradox trick, port mapping protocols, and all the hairpinning bumf. That's much nicer!

The big catch is that we currently don't have an all-IPv6 world. We have a world that's mostly IPv4, and about 33% IPv6. Those 34% are very unevenly distributed, so a particular set of peers could be 100% IPv6, 0% IPv6, or anywhere in between.

What this means, unfortunately, is that IPv6 isn't *yet* the solution to our problems. For now, it's just an extra tool in our connectivity toolbox. It'll work fantastically well with some pairs of peers, and not at all for others. If we're aiming for "connectivity no matter what", we have to also do IPv4+NAT stuff.

Meanwhile, the coexistence of IPv6 and IPv4 introduces yet another new scenario we have to account for: NAT64 devices.




So far, the NATs we've looked at have been NAT44: they translate IPv4 addresses on one side to different IPv4 addresses on the other side. NAT64, as you might guess, translates between protocols. IPv6 on the internal side of the NAT becomes IPv4 on the external side. Combined with DNS64 to translate IPv4 DNS answers into IPv6, you can present an IPv6-only network to the end device, while still giving access to the IPv4 internet.

(Incidentally, you can extend this naming scheme indefinitely. There have been some experiments with NAT46; you could deploy NAT66 if you enjoy chaos; and some RFCs use NAT444 for carrier-grade NAT.)

tailscale

involved without you being any the wiser.

But we care deeply about specific IPs and ports for our NAT and firewall traversal. What about us? If we're lucky, our device supports CLAT (Customer-side translator — from Customer XLAT). CLAT makes the OS pretend that it has direct IPv4 connectivity, using NAT64 behind the scenes to make it work out. On CLAT devices, we don't need to do anything special.

CLAT is very common on mobile devices, but very uncommon on desktops, laptops and servers. On those, we have to explicitly do the work CLAT would have done: detect the existence of a NAT64+DNS64 setup, and use it appropriately.

Detecting NAT64+DNS64 is easy: send a DNS request to `ipv4only.arpa.` That name resolves to known, constant IPv4 addresses, and only IPv4 addresses. If you get IPv6 addresses back, you know that a DNS64 did some translation to steer you to a NAT64. That lets you figure out what the NAT64 prefix is.

From there, to talk to IPv4 addresses, send IPv6 packets to `{NAT64 prefix + IPv4 address}`. Similarly, if you receive traffic from `{NAT64 prefix + IPv4 address}`, that's IPv4 traffic. Now speak STUN through the NAT64 to discover your public `ip:port` on the NAT64, and you're back to the classic NAT traversal problem — albeit with a bit more work.

Fortunately for us, this is a fairly esoteric corner case. Most v6-only networks today are mobile operators, and almost all phones support CLAT. ISPs running v6-only networks deploy CLAT on the router they give you, and again you end up none the wiser. But if you want to get those last few opportunities for connectivity, you'll have to explicitly support talking to v4-only peers from a v6-only network as well.

## Integrating it all with ICE

We're in the home stretch. We've covered stateful firewalls, simple and advanced NAT tricks, IPv4 and IPv6. So, implement all the above, and we're done!

Except, how do you figure out which tricks to use for a particular peer? How do you figure out if this is a simple stateful firewall problem, or if it's time to bust out the birthday paradox, or if you need to fiddle with NAT64 by hand? Or maybe the two of you are on the same Wi-Fi network, with no firewalls and no effort required.

Early research into NAT traversal had you precisely characterize the path between you and your peer, and deploy a specific set of workarounds to defeat that exact path. But as it turned out, network engineers and NAT box programmers have many inventive ideas, and that stops scaling very quickly. We need something that involves a bit less thinking on our part.

tailscale

signalling sessions and dialing and so forth. However, if you push past that, it also specifies a stunningly elegant algorithm for figuring out the best way to get a connection.

Ready? The algorithm is: try everything at once, and pick the best thing that works. That's it. Isn't that amazing?

Let's look at this algorithm in a bit more detail. We're going to deviate from the ICE spec here and there, so if you're trying to implement an interoperable ICE client, you should go read RFC 8445 and implement that. We'll skip all the telephony-oriented stuff to focus on the core logic, and suggest a few places where you have more degrees of freedom than the ICE spec suggests.

To communicate with a peer, we start by gathering a list of candidate endpoints for our local socket. A candidate is any `ip:port` that our peer might, perhaps, be able to use in order to speak to us. We don't need to be picky at this stage, the list should include at least:

- IPv6 `ip:ports`
- IPv4 LAN `ip:ports`
- IPv4 WAN `ip:ports` discovered by STUN (possibly via a NAT64 translator)
- IPv4 WAN `ip:port` allocated by a port mapping protocol
- Operator-provided endpoints (e.g. for statically configured port forwards)

Then, we swap candidate lists with our peer through the side channel, and start sending probe packets at each others' endpoints. Again, at this point you don't discriminate: if the peer provided you with 15 endpoints, you send "are you there?" probes to all 15 of them.

These packets are pulling double duty. Their first function is to act as the packets that open up the firewalls and pierce the NATs, like we've been doing for this entire article. But the other is to act as a health check. We're exchanging (hopefully authenticated) "ping" and "pong" packets, to check if a particular path works end to end.

Finally, after some time has passed, we pick the "best" (according to some heuristic) candidate path that was observed to work, and we're done.

The beauty of this algorithm is that if your heuristic is right, you'll always get an optimal answer. ICE has you score your candidates ahead of time (usually: LAN > WAN > WAN+NAT), but it doesn't have to be that way. Starting with v0.100.0, Tailscale switched from a hardcoded preference order to round-trip latency, which tends to result in the same LAN > WAN > WAN+NAT ordering. But unlike static ordering, we

tailscale

The ICE spec structures the protocol as a "probe phase" followed by an "okay let's communicate" phase, but there's no reason you *need* to strictly order them. In Tailscale, we upgrade connections on the fly as we discover better paths, and all connections start out with DERP preselected. That means you can use the connection immediately through the fallback path, while path discovery runs in parallel. Usually, after a few seconds, we'll have found a better path, and your connection transparently upgrades to it.

One thing to be wary of is asymmetric paths. ICE goes to some effort to ensure that both peers have picked the same network path, so that there's definite bidirectional packet flow to keep all the NATs and firewalls open. You don't have to go to the same effort, but you *do* have to ensure that there's bidirectional traffic along all paths you're using. That can be as simple as continuing to send ping/pong probes periodically.

To be really robust, you also need to detect that your currently selected path has failed (say, because maintenance caused your NAT's state to get dumped on the floor), and downgrade to another path. You can do this by continuing to probe all possible paths and keep a set of "warm" fallbacks ready to go, but downgrades are rare enough that it's probably more efficient to fall all the way back to your relay of last resort, then restart path discovery.

Finally, we should mention security. Throughout this article, I've assumed that the "upper layer" protocol you'll be running over this connection brings its own security (QUIC has TLS certs, WireGuard has its own public keys...). If that's not the case, you absolutely need to bring your own. Once you're dynamically switching paths at runtime, IP-based security becomes meaningless (not that it was worth much in the first place), and you *must* have at least end-to-end authentication.

If you have security for your upper layer, strictly speaking it's okay if your ping/pong probes are spoofable. The worst that can happen is that an attacker can persuade you to relay your traffic through them. In the presence of e2e security, that's not a *huge* deal (although obviously it depends on your threat model). But for good measure, you might as well authenticate and encrypt the path discovery packets as well. Consult your local application security engineer for how to do that safely.

## Concluding our connectivity chat

At last, we're done. If you implement all of the above, you'll have state of the art NAT traversal software that can get direct connections established in the widest possible array of situations. And you'll have your relay network to pick up the slack when traversal fails, as it likely will for a long tail of cases.

This is all quite complicated! It's one of those problems that's fun to explore, but quite fiddly to get right, especially if you start chasing the long tail of opportunities for just

tailscale

get to explore the exciting and relatively under-explored world of peer-to-peer applications. So many interesting ideas for decentralized software fall at the first hurdle, when it turns out that talking to each other on the internet is harder than expected. But now you know how to get past that, so go build cool stuff!

**Here's a parting "TL;DR" recap:** For robust NAT traversal, you need the following ingredients:

- A UDP-based protocol to augment
- Direct access to a socket in your program
- A communication side channel with your peers
- A couple of STUN servers
- A network of fallback relays (optional, but highly recommended)

Then, you need to:

- Enumerate all the `ip:ports` for your socket on your directly connected interfaces
- Query STUN servers to discover WAN `ip:ports` and the "difficulty" of your NAT, if any
- Try using the port mapping protocols to find more WAN `ip:ports`
- Check for NAT64 and discover a WAN `ip:port` through that as well, if applicable
- Exchange all those `ip:ports` with your peer through your side channel, along with some cryptographic keys to secure everything.
- Begin communicating with your peer through fallback relays (optional, for quick connection establishment)
- Probe all of your peer's `ip:ports` for connectivity and if necessary/desired, also execute birthday attacks to get through harder NATs
- As you discover connectivity paths that are better than the one you're currently using, transparently upgrade away from the previous paths.
- If the active path stops working, downgrade as needed to maintain connectivity.
- Make sure everything is encrypted and authenticated end-to-end.



tailscale

## Subscribe to Tailscale's blog

We have a deep commitment to keeping your data safe.

Enter your email Submit

Too much email? RSS X

# More articles

Apr 11, 2024

### Remotely access Home Assistant via Tailscale for free

Alex Kretzschmar

Mar 29, 2024

### About the Tailscale.com outage on March 7, 2024

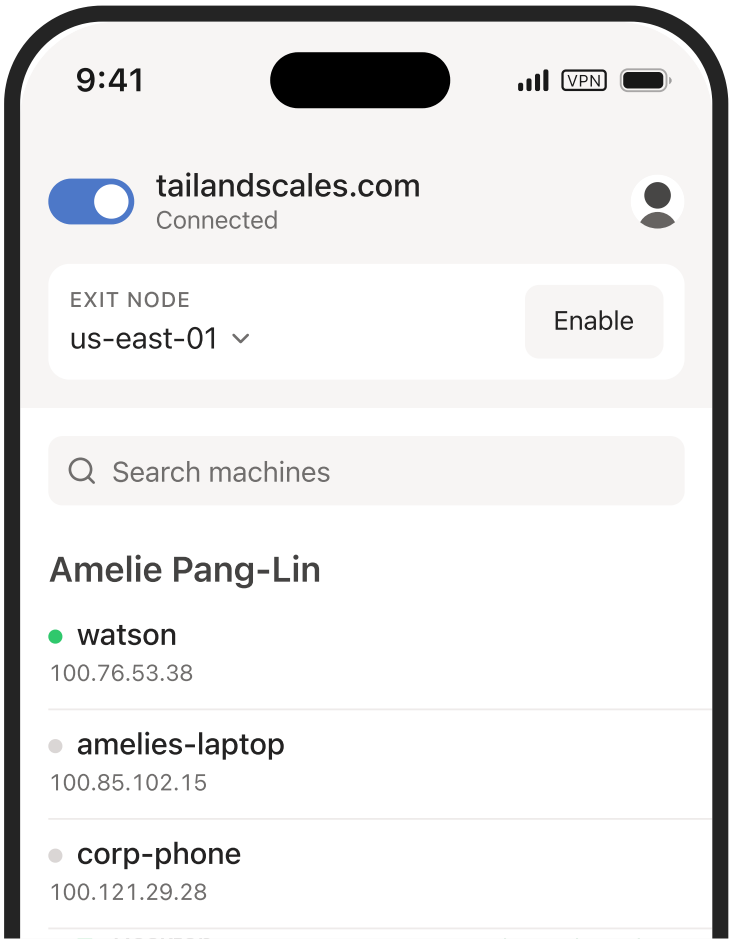
tailscale

Mar 22, 2024

## Tailscale SSH is now Generally Available

Sam Linville

tailscale




MERCURY instacart Retool

duolingo mercari

Product
- How it works
- Pricing
- Integrations
- Features

Use Cases
- Business VPN
- Remote Access
- Site-to-Site Networking
- Homelab

Resources
- Blog
- Events & Webinars

tailscale

Company
Careers
Press

Support
Sales
Security
Legal
Open Source
Changelog

SSH keys
Docker SSH
DevSecOps
Multicloud
NAT Traversal
MagicDNS
PAM
PoLP
All articles

tailscale

Terms of Service

Privacy Policy

WireGuard is a registered trademark of Jason A. Donenfeld.