Debian, IPv6, and Hurricane Electric HOWTO

Back to top page

Table of Contents

Assumptions

- You are familiar with the basics of IPv4 and IPv6. (In particular, you know how to expand "`2001:db8::89ab`" to the 128-bit address it represents, and you know what CIDR masks like `/8` and `/64` mean.)
- You want to add IPv6 connectivity to your local network with IPv4 Internet connectivity.
- You have a Linux system with a statically-assigned public IPv4 address, with no additional firewalls or NATs between you and the Internet. (Or, if you have them, they're willing to pass IPv4 protocol 41 to your system.)
  - You might be using this system as your existing IPv4 router to the local network, with or without NAT. But it could just be some random host on your local network.
- Your system is running Debian Lenny (5.0) or greater.
  - Odds are good that older Debian releases still work, although the instructions might need adaptation.
  - Other Debian derivatives will probably work, too. For instance, these instructions would probably work with Ubuntu Hardy Heron and later, although you might need to fiddle with NetworkManager.
  - Other Linux distributions won't use `apt-get`, might not use `/etc/network/interfaces`, and so on. The radvd and BIND configuration steps are still pretty similar.

Background

On February 3rd, 2011, IANA assigned the last remaining blocks of IPv4 addresses to the regional authorities: ARIN, LACNIC, RIPE, AfriNIC, and APNIC. Since then, most of these regional authorities have effectively run out of IPv4 addresses. This means that new businesses can't put their employees' computers on the Internet, that ISPs can't grow to cover new customers, and so on.

The long-term solution, IPv6, was hammered out more than a decade ago, and most of the standards have been settled for many years now. However, a large body of routing equipment lacks IPv6 support: big gear from vendors like Cisco and Juniper tends to be a big capital expense, so a lot of aging deployments predate their manufacturers' IPv6 support, or lacks IPv6 features that took a few years to standardize; meanwhile, home routers are built on a shoestring budget and often have buggy implementations of *IPv4*, much less IPv6.

It's now clear that the future of the Internet will contain a transition period where patchy islands of IPv6 support are connected to the IPv6 Internet over IPv4 tunnels, and this will last until (a) all this legacy routing equipment (mostly ISP gear, not backbone gear) is replaced, (b) the new equipment is configured correctly for IPv4/IPv6 dual stack operation, and (c) manufacturers of consumer gear start treating IPv6 support as a marketable feature. At this point it's pretty clear how the Internet's future will settle down: IPv4 addresses will be too rare to hand out to clients, so clients in the near future will suffer from a mess of NAT technologies for accessing the traditional IPv4 Internet, with client-side IPv6 rollout picking up momentum in fits and starts as ISPs upgrade their equipment and customers replace their local routing gear with next year's model. Meanwhile, servers will have a strong incentive to go dual-stack to support Asia/Pacific clients, with IPv4 addresses more expensive but still available until ISPs *slowly* begin turning off the lights for the IPv4 Internet.

For configuring an IPv6-over-IPv4 tunnel, there are a number of options:

- 6to4 lets any IPv4 user reach the IPv6 backbone by wrapping IPv6 packets in IPv4 (protocol 41, a.k.a. "`6in4`", **not** synonymous with "`6to4`") and sending them to the 6to4 anycast address, `192.88.99.1`. These anycast routers are operated as an unfunded public service, mostly by universities. (The performance is as good as it sounds.) The return paths aren't always reliable, either.
- 6rd is similar to 6to4 (it uses protocol 41), but your ISP runs the backbone-side router. You get better latency (closer) and performance (monetary stake) than 6to4. (Comcast is experimenting with this, but the customer side needs some Linux kernel support that isn't widely deployed yet.)
- Teredo has conceptual similarities to 6to4, but it tunnels IPv6 packets through UDP/IPv4, which is easier to punch through some IPv4 NAT devices. Microsoft likes this one; they run some Teredo endpoints, and Windows 7 has Teredo enabled by default. I'm not sure how plentiful Teredo endpoints are, but they probably suffer from most of 6to4's issues.
- You can sign up with a tunnel broker and use a real point-to-point tunnel. This requires a bit of initial legwork, but can provide the latency/performance benefits of 6rd without waiting for your ISP to pull their head out. Most of these offer protocol 41, but some have additional options (e.g. TSP or AYIYA). Protocol 41 is a perfectly reasonable way to reach the backbone, though, if your gear can route it.
- There are a maddeningly large number of other protocols out there that I haven't mentioned.

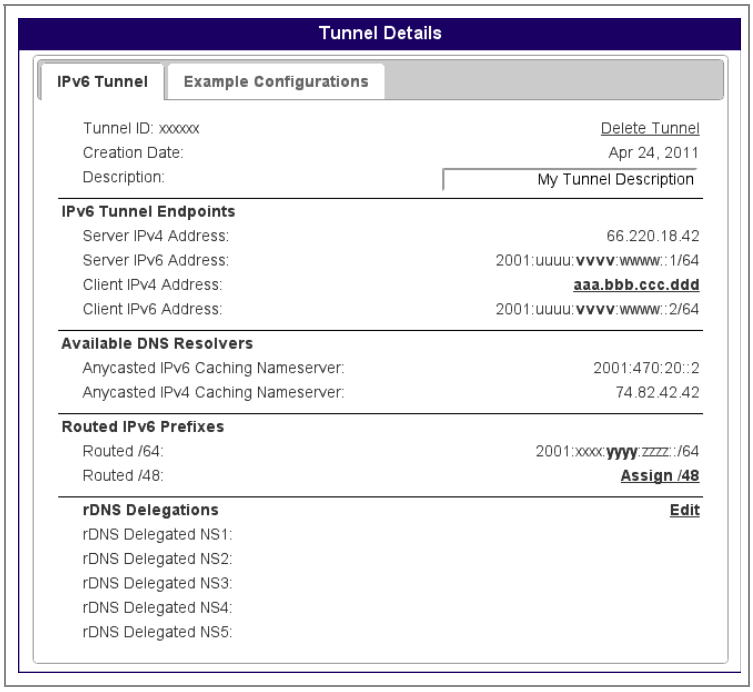
If you just have one computer and you're not using it as a server, then 6to4 and Teredo are reasonable choices for reaching the handful of single-stack IPv6 hosts out there. However, if you're running a server, you'll want a reliable return path and a nearby tunnel endpoint, so 6rd or a nearby tunnel broker is more appropriate. And if you're routing IPv6 to multiple devices on your local network, then a tunnel broker is definitely the better option because it gives you more control over your network deployment.

Hurricane Electric has built quite the reputation on their IPv6 services, so I decided to use their tunnel broker service. SixXS is an alternative.

Creating the tunnel

Visit HE's tunnel broker website (www.tunnelbroker.net) and **fill out the sign-up form**. They'll want to know some basic contact information, none of which is particularly unusual by network routing standards (full name, street address, phone #, e-mail address), and they'll send you your initial password to confirm that you own the e-mail address. Log in to their website, change the password to something secure (i.e. not sent over cleartext SMTP), then follow the "Create Regular Tunnel" link. Enter the public IPv4 address of your Debian server, and select the HE-side tunnel endpoint with the lowest latency to your Debian server. (In general, geographic distance is a good stand-in for network latency, so there's no need to fret with traceroute if you know your local geography.)

IIRC, from there the "Create Tunnel" button directly sends you to the "Tunnel Details" page for your new tunnel. If not, navigate to it:



A brief explanation of what these represent:

- "Server IPv4 Address" (`66.220.18.42` in this example) is the address of their router on the IPv4 Internet. This is the remote end of our IPv6-over-IPv4 tunnel.
- "Client IPv4 Address" (`aaa.bbb.ccc.ddd`) is the existing address of your server on the IPv4 Internet. This will be the local end of our IPv6-over-IPv4 tunnel.
- "Server IPv6 Address" (`2001:uuuu:vvvv:wwww::1`) is the tunnel-side IPv6 address of their router. We'll set this as our gateway for IPv6 traffic.
- "Client IPv6 Address" (`2001:uuuu:vvvv:wwww::2`) is the tunnel-side IPv6 address of your server. This will almost never get used.
- "Routed /64" (`2001:xxxx:yyyy:zzzz::/64`) is the slice of the public IPv6 space that has been assigned to your local network. Packets with destination addresses in this range will be sent to your server for further routing.
  - In my case: `xxxx` = `uuuu`, `yyyy` = `vvvv` + 1, `zzzz` = `wwww`. I recommend you treat any such pattern as an implementation detail, subject to change when future tunnels are assigned.
  - In examples, we're going to make `2001:xxxx:yyyy:zzzz::1` be your server address. Don't confuse it with `2001:uuuu:vvvv:wwww::1`, even though the difference might be only one hex digit.

Hosts on your local network will use your server as their IPv6 gateway. (Presumably your server is already their IPv4 gateway, but that's not strictly necessary.) Your server puts IPv6 packets in the tunnel by wrapping them with IPv4 envelopes — IPv4 protocol 41 ("`ipv6`"), which adds 20 bytes to each packet — and then sending the tunneled packets to `66.220.18.42` over your existing IPv4 route(s). At HE's router, the tunneled packets are removed from their envelopes and injected into the IPv6 backbone as if they'd always been there. In the other direction, packets on the backbone bound for `2001:xxxx:yyyy:zzzz::/64` are routed back to HE's network in the region you requested — presumably via BGP route announcement, but that's HE's problem, not yours — then HE's router encapsulates them in IPv4 and tunnels them back to your server's IPv4 address (`aaa.bbb.ccc.ddd`). Your server is then responsible for decapsulating them and routing them within your local network.

It's important to keep in mind that your server will be a member of two *different* IPv6 netblocks, both with public routing. If your local network already has

a public IPv4 assignment, then this is probably obvious to you, but if you're used to the NAT mindset you may need to break some old habits of thought here.

Something that may throw you for a loop: the tunnel itself has an entire /64 delegated to it, even though it only has 2 addresses assigned. The mind boggles. I'm not quite sure why they felt the need to use routable addresses instead of `fe80::/64` (or something in `fd00::/8`). Then again, given that there are 281 trillion /64s in `2001::/16` alone... it's not like we're about to run out. (~~And I guess it would be slightly annoying for `fe80::something` to show up in traceroutes. Now that I think about it, the ICMPv6 Time Exceeded errors from HE's router would come from the backbone-side address, not the tunnel-side address. So traceroute isn't a good excuse for handing out a routable assignment for the tunnel.~~ Duh, the world would still see ICMPv6 Time Exceeded errors from the tunnel side of your own gateway. So, yeah, having a routable tunnel address is probably for the best.)

Debian to HE to Internet

Fire up your favorite command-line text editor as root using `sudo`, pull up **`/etc/network/interfaces`**, and add a definition for the IPv6-over-IPv4 tunnel:

```
auto hurricane0
iface hurricane0 inet6 v4tunnel
  address 2001:uuuu:vvvv:wwww::2
  netmask 64
  endpoint 66.220.18.42
  local aaa.bbb.ccc.ddd
  gateway 2001:uuuu:vvvv:wwww::1
  ttl 255
```

Now save the file and run "**`sudo ifup hurricane0`**". This will bring up your end of the tunnel. Try reaching `ipv6.google.com` to verify connectivity:

```
me@my-server:~$ wget -O/dev/null http://ipv6.google.com/
--2015-03-10 21:28:58--  http://ipv6.google.com/
Resolving ipv6.google.com (ipv6.google.com)... 2607:f8b0:4000:804::1002
Connecting to ipv6.google.com (ipv6.google.com)|2607:f8b0:4000:804::1002|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [text/html]
Saving to: '/dev/null'

  [ <=>                                   ] 17,638      --.-K/s   in 0.04s

2015-03-10 21:28:59 (395 KB/s) - '/dev/null' saved [17638]
```

That was easy, wasn't it?

Debian to LAN

Fire up your editor and pull up **`/etc/network/interfaces`** one more time. Find your LAN interface, and add an inet6 stanza:

```
# Existing
auto eth-lan
iface eth-lan inet static
  address 192.168.0.1
  netmask 255.255.255.0
  broadcast 192.168.0.255

# Added
iface eth-lan inet6 static
  address 2001:xxxx:yyyy:zzzz::1
  netmask 64

# Incidentally, ifrename(8) plus iftab(5) rocks.
```

Modern versions of `ifup(8)` should have very little trouble with multiple stanzas like this.

Now save your changes. You can apply them without disruption by running "**`/sbin/ip addr add 2001:xxxx:yyyy:zzzz::1/64 dev eth-lan`**".

LAN to Debian to HE to Internet

First things first: you need to enable IPv6 forwarding on your kernel. Run "**`/sbin/sysctl -n net.ipv6.conf.all.forwarding`**" and verify that it's set to "`1`". If it's not, you'll need to add it to **`/etc/sysctl.conf`**; it might already exist as a comment, preceded by "`# Uncomment the next line to enable packet forwarding for IPv6`". Uncomment the line if it exists, or simply add "**`net.ipv6.conf.all.forwarding = 1`**" if it's missing, then save the file. Be sure to apply it with "**`sudo /sbin/sysctl -q -p`**".

Now that routing's enabled, any host behind your server can talk to the Internet… if only it had an address. You could configure your hosts manually, but we're not going to do that.

Most modern versions of Linux and Windows can autoconfigure themselves if they receive a `Router Advertisement` packet via the ICMPv6 Neighbor Discovery protocol. The standard way of sending a `Router Advertisement` from Linux is `radvd(8)`, the "Router ADVertisement Daemon". Compared to DHCP, this is much nicer: your server maintains no state, each client only needs to remember the last advertisement it saw, and all clients on a LAN segment can configure themselves at the same time from a single broadcast packet. The overhead is so much lower than DHCP that Neighbor Discovery is usually deployed as a "heartbeat" protocol: if clients stop seeing `Router Advertisement` packets from their router, they assume the router is dead and look for alternative paths to the Internet.

To get started with stateless autoconfiguration, install the `radvd` package on your Debian server with "**`sudo apt-get install radvd`**".

The server won't automatically start, because you need to configure it first. To fix that, create **`/etc/radvd.conf`** with contents similar to the following:

```
interface eth-lan {
  AdvSendAdvert on;
  AdvLinkMTU 1480;
  MinRtrAdvInterval 60;
  MaxRtrAdvInterval 180;
  prefix 2001:xxxx:yyyy:zzzz::1/64 {
    AdvOnLink on;
    AdvRouterAddr on;
    AdvPreferredLifetime 600;
    AdvValidLifetime 3600;
  };
  route ::/0 {
  };
};
```

This is what the options mean:

- "`AdvSendAdvert on`" means that `radvd` sends `Router Advertisement` broadcasts on a timer. If this were off, hosts on the LAN would need to explicitly send `Router Solicitation` broadcasts on a regular basis.
- "`AdvLinkMTU 1480`" means "FYI, packets on this Ethernet segment are capped at 1480 bytes". Ideally there'd be a way to say "FYI, packets *going through this router* are capped at 1480 bytes", but there isn't, and (for *most* traffic) speeding up PMTU discovery is a bigger benefit than the drawback of unnecessarily shrinking packets on the local segment by a mere 20 bytes.
- "`MinRtrAdvInterval 60`" and "`MaxRtrAdvInterval 180`" means that radvd should send `Router Advertisement` broadcasts *at least* once every 3 minutes, but *at most* once per minute. Individual clients can send `Router Solicitation` requests to trigger an early broadcast.
- "`prefix 2001:xxxx:yyyy:zzzz::1/64`" tells clients that they can create an autoconfigured address by sticking `2001:xxxx:yyyy:zzzz` in front of a 64-bit unique identifier (generally a MAC address in EUI-64 form). This option also tells the clients about the router's public IP (see `AdvRouterAddr` below).
- "`AdvOnLink on`" means "Yes, you can safely assume that everyone in `2001:xxxx:yyyy:zzzz::/64` is on the same Ethernet segment as you, so there's no need to go through me, your friendly but overworked router". Obviously, if your local network spans multiple Ethernet segments, change this to "`off`".
- "`AdvRouterAddr on`" means "You know that prefix I just told you about? If you ignore the /64, it's also my public IPv6 address. If you'd like, feel free to talk to me using that instead of my `fe80::` address". This comes from the Mobile IPv6 spec, but you don't need to set up a full Mobile IPv6 deployment to use it.
- "`AdvPreferredLifetime 600`" and "`AdvValidLifetime 3600`" means "Hey, if you don't hear from me for 10 minutes, form a search party, and if you don't hear from me for an hour, call off the search because I'm either kidnapped or dead". The clients will gradually stop using their autoconfigured address in this prefix (first deprecating it, then deleting it).
- "`route ::/0`" tells clients that they can route traffic to the rest of the IPv6 Internet through this router. Since we didn't specify `AdvRouteLifetime`, the route is valid for `3*MaxRtrAdvInterval` = 9 minutes at a time, constantly refreshed by `Router Advertisement` broadcasts.

This configuration doesn't tell the clients how to resolve DNS, on the assumption that one or more recursive DNS servers were already configured for their respective IPv4 stacks. If your server is already running an appropriately configured BIND (i.e. it listens on IPv6 and hosts on `2001:xxxx:yyyy:zzzz::/64` are allowed to perform recursive queries), then simply add "`RDNSS 2001:xxxx:yyyy:zzzz::1 {};`". You could also point clients at HE's anycast resolver ("`RDNSS 2001:470:20::2 {};`"), or at Google's anycast resolvers ("`RDNSS 2001:4860:4860::8888 2001:4860:4860::8844 {};`").

Now that the daemon is configured, start it with "**`sudo /etc/init.d/radvd start`**". It will immediately begin sending `Router Advertisement` broadcasts, and hosts supporting Neighbor Discovery will autoconfigure their IPv6 stacks as soon as they see it. Once you start the daemon, an IPv6 LAN can go from zero to functional in just a few seconds.

Now that the local network has IPv6 connectivity, you might try visiting http://test-ipv6.com/ in a Javascript-capable web browser. If your recursive DNS resolver is sane, a score of 10/10 should be easy to obtain. (If it's not sane, BIND isn't too hard to set up, and there are other options in APT if you just need a recursive resolver that doesn't serve zones. Or use Google's servers.)

Resolving Names to IPv6 Addresses

If you run BIND on your Debian server, it's pretty trivial to add IPv6 records. Assuming you own `example.com`, all you need to do is pull up your zone file

for `example.com` (poke through **`/etc/bind/named.conf`** and its includes if you forgot where you keep your zone file) and then start adding `AAAA` records for every hostname you care about. For example:

```
$TTL 3600
@          SOA     my-server admin 2011042501 14400 3600 259200 3600
           NS      ns1
           NS      ns2
           NS      ns3
ns1        A       aaa.bbb.ccc.ddd
           AAAA    2001:xxxx:yyyy:zzzz::1  ; Add me
ns2        A       aaa.bbb.ccc.eee
ns3        A       fff.ggg.hhh.iii
www        CNAME   my-server
my-server A        aaa.bbb.ccc.ddd
           AAAA    2001:xxxx:yyyy:zzzz::1  ; Add me
```

Don't forget to **update the serial number** on your `SOA` record before running "**`sudo rndc reload example.com; sudo rndc notify example.com`**".

If you use a different DNS server, or you use hosted DNS, it should still be pretty easy: the zone file syntax is defined by the DNS RFCs, so zone files are pretty close to universal; and even among the hosted DNS providers that don't allow zone files and force you to choose record types from a drop-down list, most of them permit `AAAA` records these days.

To verify that your zone changes have propagated, try resolving your `AAAA` record using one of your slaves:

```
me@my-server:~$ host -t AAAA my-server.example.com. ns2.example.com.
Using domain server:
Name: ns2.example.com.
Address: aaa.bbb.ccc.eee#53
Aliases:

my-server.example.com. has IPv6 address 2001:xxxx:yyyy:zzzz::1
```

> **→IMPORTANT!**
>
> If the host you're enabling IPv6 on is also your DNS server, then you need to add IPv6 glue records so that IPv6-only hosts can reach it. As a reminder, glue records are required if you have DNS servers that (a) are authoritative for `example.com`, and also (b) have names that live in `example.com`. (For example, if one of the `NS` records for `example.com` is `ns1.example.com`, then `ns1.example.com` needs a glue record.)
>
> The process for updating your glue records will depend strongly on which domain registrar you use.
>
> For instance, if you use Gandi.net, you should log in, follow the link for your domain, and follow the link for "Glue record management". This will give you the UI for adding and editing glue records. Gandi simply accepts a line-by-line list of IP addresses; IPv4 and IPv6 are both acceptable. Each line will result in an `A` or `AAAA` glue record, as appropriate.

Reversing IPv6 Addresses to Names

This is slightly trickier than forward resolution — and if you're using a hosted DNS service, they might not support it.

Assuming again that you run BIND on your Debian server, you need to configure BIND to claim itself as authoritative for your prefix's reverse zone, and then you need to ask HE to delegate the reverse zone to your server and its slaves.

First, you need to **figure out the name of your reverse zone**. This is accomplished by taking your prefix (`2001:xxxx:yyyy:zzzz`), adding in the missing zeroes (if any) to each component (e.g. `2001:xxx:y:zz` becomes `2001:0xxx:000y:00zz`), and then reversing the nibble digits, sticking a dot between each pair of them, and appending "`ip6.arpa`" to the end of that mess. The end result will look like "`z.z.z.z.y.y.y.y.x.x.x.x.1.0.0.2.ip6.arpa`" (keeping in mind that `z.z.z.z` is digit-for-digit backwards compared to `zzzz` and so on). Stick that in a convenient place (GUI clipboard, Vim register, Emacs kill ring, yadda) because you won't want to type that more than once.

Now edit **`/etc/bind/named.conf`** (or the appropriate include file, depending on how you've structured your BIND configuration) and find your existing zone declarations. Add a master zone declaration for your reverse zone; something similar to the following should suffice:

```
zone "z.z.z.z.y.y.y.y.x.x.x.x.1.0.0.2.ip6.arpa" {
  type master;
  file "rev/2001-xxxx-yyyy-zzzz";
};
```

Before you save that, create the new reverse zone file you just declared. It should look something like this:

```
$TTL 3600
```

```
@   SOA my-server.example.com. admin.example.com. 2011042501 14400 3600 259200 3600
    NS  ns1.example.com.
    NS  ns2.example.com.
    NS  ns3.example.com.
1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0 PTR my-server.example.com.
```

Now save your new reverse zone file, save your BIND configuration, and run "`sudo rndc reconfig`" to load the zone in BIND.

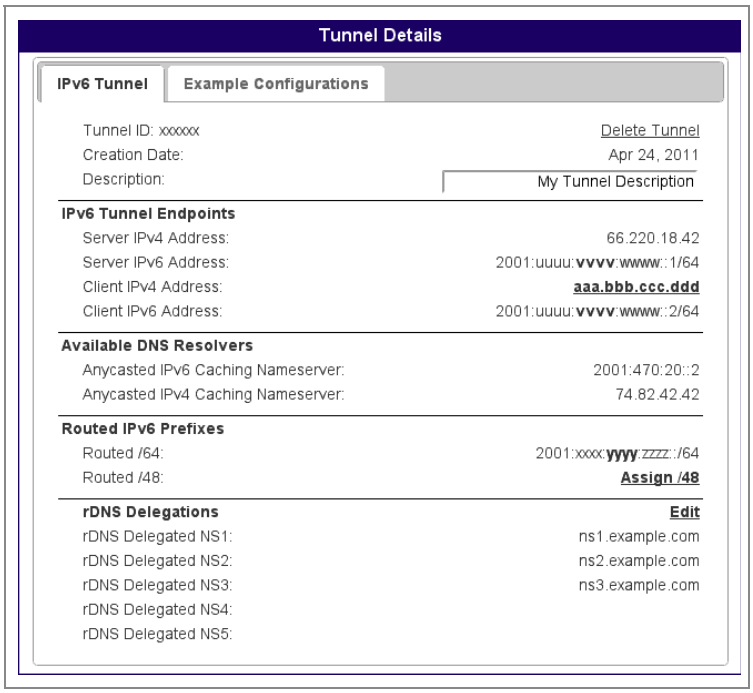
Verify that you can reverse resolve your IP:

```
me@my-server:~$ host 2001:xxxx:yyyy:zzzz::1 ::1
Using domain server:
Name: ::1
Address: ::1#53
Aliases:

1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.z.z.z.z.y.y.y.y.x.x.x.x.1.0.0.2.ip6.arpa domain name pointer my-server.example.com.
```

Don't forget to configure your slaves and to add `NS` records to the zone for them.

Now that your reverse zone is ready, the last step is to **tell Hurricane Electric** to use it. Go back to the Tunnel Details page (you may need to log in again), then find the "rDNS Delegations" section, which defaults to being blank. Click the "Edit" button and you'll see an interface for specifying which DNS servers will be serving the zone. Specify your Debian server's name as NS1, and list the names of your slaves as NS2 through NS5. Go ahead and save your configuration; it should look something like this:



Once the delegation records propagate (should be quick), you will be able to reverse-resolve your IP using any recursive resolver. For instance, using one of Google's public recursive resolvers:

```
me@my-server:~$ host 2001:xxxx:yyyy:zzzz::1 8.8.8.8
Using domain server:
Name: 8.8.8.8
Address: 8.8.8.8#53
Aliases:

1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.z.z.z.z.y.y.y.y.x.x.x.x.1.0.0.2.ip6.arpa domain name pointer my-server.example.com.
```

License