Configuring tunnels with iproute2

Simone Piunno

Deep Space 6

simone@deepspace6.net

1. iproute2

iproute2 is a package for advanced network management under linux. In practice, it is composed of a bunch of small utilities to dinamically configure the kernel by means of rtnetlink sockets - a modern and powerful interface for the configuration of the networking stack implemented by Alexey Kuznetsov starting from the 2.2 kernel series.

The most interesting feature of **iproute2** is that it replaces with a single integrated and organic command all the functionalities we were used to find in **ifconfig**, **arp**, **route** and **iptunnel** (and it even adds some more!).

Nowadays **iproute2** is installed by default on most major distributions, even if their initialization scripts are still built on commands from the old **net-tools** package (e.g. **ifconfig** or **iptunnel** - the latter is actually deprecated). If your distribution doesn't include this important package, you can always download it from ftpsite and compile it yourself.

As the time of this writing, the worst defect of **iproute2** is a relative lack of documentation, partially compensated by the fact that the syntax of the **ip** command is very easy and similar to the english language. We believe that people used to **ifconfig** and **route** shouldn't encounter any problem using **ip** and that they will feel at home in a macommander of hours. In this document we will suppose that the reader has already a good knowledge of basic networking concepts and has used **ifconfig** and **route** in the past.

2. Introduction to tunnels

Let's imagine two Internet nodes wanting to exchange data traffic over a protocol different from IPv4 or directed to a private LAN using non-globally-valid IP addresses. This problem is typically solved using a virtual point-to-point connection between the two nodes and we call this configuration a *tunnel*.

You can think to every packet traveling over the network like it was an envelope with a few bits inside and the sender's and receiver's addresses written on. Tunnels simply hide this envelope inside an additional one, with different sender and receiver, effectively diverting the packet's trip. When the packet

arrives to the external receiver (the one written on the external envelope), the external envelope is removed and thrown away, so that the packet can continue its travel to the real destinantion.

The two nodes putting and removing the additional envelope are called *endpoints* and need to have a known IPv4 address. This is why tunnels generally don't work when traversing a network address translation (NAT). Moreover, if the tunnel is built throuh a firewall, the latter must be configured ad hoc to permit this kind of traffic.

A typical tunnel usage is connecting two IPv6 nodes through an IPv4-only network. The two nodes can build an IPv6-in-IPv4 tunnel pretending to have a real direct point-to-point IPv6 connection, and this way they can link together two IPv6 islands (6bone works this way, a web of tunnels). Tunnels for IPv6-over-IPv4 transport come in two different flawors: automatic RFC2373 and manually configured. In this document we will talk only of the latter type.

3. Creating tunnels

Creating tunnels with **iproute2** is very easy. First of all you need a name for your tunnel. If you choose to name it foo then you can create the tunnel with the command:

```
ip tunnel add foo mode sit remote 192.168.1.42
```

This way, you created a sit (IPv6-in-IPv4) tunnel with a remote endpoint at the IP address 192.168.1.42. Notice that we have not specified which IP address to use for the local side of the tunnel, which interface, and so on. The result can be viewed with the command **ip tunnel show**:

```
# ip tunnel show
sit0: ipv6/ip  remote any  local any  ttl 64  nopmtudisc
foo: ipv6/ip  remote 192.168.1.42  local any  ttl inherit
```

Our tunnel is the one in the 2nd row. Now we can also ask a list of all available interfaces, regardless if they are real network adapters or software simulations:

```
# ip link show
1: lo: <loopback,up> mtu 16436 qdisc noqueue
  link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <broadcast,multicast,up> mtu 1500 qdisc pfifo_fast qlen 100
  link/ether 00:48:54:1b:25:30 brd ff:ff:ff:ff:ff:ff
4: sit0@none: <noarp> mtu 1480 qdisc noop
  link/sit 0.0.0.0 brd 0.0.0.0
6: foo@none: <pointopoint,noarp> mtu 1480 qdisc noop
  link/sit 0.0.0.0 peer 192.168.1.42
```

The fact that should get your attention is that while lo and eth0 are marked as being `up`, our tunnel is not. To double check, the good old **ifconfig** says only:

```
# ifconfig
eth0    Link encap:Ethernet  HWaddr 00:48:54:1b:25:30
        inet addr:192.168.0.1  Bcast:192.168.0.255  Mask:255.255.255.0
        inet6 addr: fe80::248:54ff:fe1b:2530/10 Scope:Link
        UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
        RX packets:0 errors:0 dropped:0 overruns:0 frame:0
        TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:100
        RX bytes:0 (0.0 b)  TX bytes:528 (528.0 b)
        Interrupt:9 Base address:0x5000

lo      Link Encap:Local Loopback
        inet addr:127.0.0.1  Mask:255.0.0.0
        inet6 addr: ::1/128 scope:host
        UP LOOPBACK RUNNING  MTU:16436  Metric:1
        RX packets:35402 errors:0 dropped:0 overruns:0 frame:0
        TX packets:35402 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:0
        RX bytes:3433996 (3.2 mb)  TX bytes:3433996 (3.2 mb)
```

So we must remember that the **ip link** command shows all available interfaces, regardless of them being activated or not. To activate foo, we use the command:

```
ip link set foo up
```

and to deactivate it:

```
ip link set foo down
```

To completely discard our tunnel we use:

```
ip tunnel del foo
```

4. Special tunnels

In the previous paragraph, we've seen how to build an IPv6-in-IPv4 tunnel, now we'll examine a few different situations.

4.1. GRE tunnels

If you don't need IPv6 but for example you want to carry normal IPv4 traffic through a non-cooperating transit network, then you'd better use *`mode gre`* instead of *`mode sit`*. For example:

```
# ip tunnel add foo4 mode gre remote 192.168.1.42
# ip tunnel show
gre0: gre/ip  remote any  local any  ttl inherit  nopmtudisc
foo4: gre/ip  remote 192.168.1.42  local any  ttl inherit
# ip link show
1: lo: <loopback,up> mtu 16436 qdisc noqueue
  link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <broadcast,multicast,up> mtu 1500 qdisc pfifo_fast qlen 100
  link/ether 00:48:54:1b:25:30 brd ff:ff:ff:ff:ff:ff
7: gre0@none: <noarp> mtu 1476 qdisc noop
  link/gre 0.0.0.0 brd 0.0.0.0
9: foo4@none: <pointopoint,noarp> mtu 1476 qdisc noop
  link/gre 0.0.0.0 peer 192.168.1.42
```

GRE RFC2784 is a particular tunnelling protocol supported by Cisco routers which is capable to carry different protocols over IPv4. There's another kind of tunnels implemented by linux: `ipip`. The latter is also useful for IPv4-in-IPv4 encapsulation, but it's implemented only by linux and does only unicast IP over IP (so you can't transport for example IPX or broadcasts). In general, GRE is better.

4.2. Explicit local endpoint

Even if the kernel is smart enough to choose for you, it could be a good idea to explicitly force the local IP address and interface we're going to use for tunneling. To do that, we can use the *`local`* and *`dev`* parameters:

```
# ip tunnel add foo mode sit local 192.168.0.1 remote 192.168.1.42 dev eth0
# ip tunnel show
sit0: ipv6/ip  remote any  local any  ttl 64  nopmtudisc
foo: ipv6/ip  remote 192.168.1.42  local 192.168.0.1  dev eth0  ttl inherit
# ip link show
1: lo: <loopback,up> mtu 16436 qdisc noqueue
 link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <broadcast,multicast,up> mtu 1500 qdisc pfifo_fast qlen 100
 link/ether 00:48:54:1b:25:30 brd ff:ff:ff:ff:ff:ff
4: sit0@none: <noarp> mtu 1480 qdisc noop
 link/sit 0.0.0.0 brd 0.0.0.0
11: foo@eth0: <pointopoint,noarp> mtu 1480 qdisc noop
 link/sit 192.168.0.1 peer 192.168.1.42
```

Please notice that now the interface is labeled as foo@eth0, to remind us where the tunnel has been explicitly connected.

4.3. Time-to-live

When using tunnels, creating accidental loops in the network it's easy. To limit the problem, it's fundamental to generate packets with a low TTL value. Initial TTL can be specified by the `ttl` parameter in **ip tunnel add**. The default value is inherited from the network interface the tunnel is associated to. IANA suggests using 64 for TTL.

5. Assigning an IP address to the interface

Like any other network interface, tunnels can have one or more addresses assigned to them.

5.1. Main address

Assigning the main address is straightforward:

```
ip addr add 3ffe:9001:210:3::42/64 dev foo
ip addr add 192.168.0.2/24 dev foo4
ip addr add 10.20.30.40/8 dev eth0
```

The number immediately following the slash is to suggest to the kernel the network prefix we prefer, useful to automatically compute broadcast address and netmask on IPv4 LANs (this is called CIDR notation). However, tunnels are point-to-point interfaces and this number is then ignored.

Note: to be able to assign an IP address to an interface, first you need to activate the interface using:

```
ip link set interfacename up
```

To remove an address from an interface, you can obviously use **del** instead of **add**:

```
ip addr del 3ffe:9001:210:3::42/64 dev foo
ip addr del 192.168.0.2/24 dev foo4
```

We can even ask for a list of all the IP addresses in use on our server:

```
# ip addr show
1: lo: <LOOPBACK,UP> mtu 16436 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 brd 127.255.255.255 scope host lo
    inet6 ::1/128 scope host
2: eth0: <BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast qlen 100
    link/ether 00:48:54:1b:25:30 brd ff:ff:ff:ff:ff:ff
```

```
    inet 192.168.0.1/24 brd 192.168.0.255 scope global eth0
    inet6 fe80::248:54ff:fe1b:2530/10 scope link
4: sit0@NONE: <NOARP> mtu 1480 qdisc noop
    link/sit 0.0.0.0 brd 0.0.0.0
5: foo@NONE: <POINTOPOINT,NOARP> mtu 1480 qdisc noop
    link/sit 0.0.0.0 peer 192.168.1.42
    inet6 3ffe:9001:210:3::42/64 scope global
    inet6 fe80::c0a8:1/10 scope link
```

5.2. Aliasing

When using multiple addresses on a single interface, people used to **ifconfig** will be surprised noting that multiple **ip addr add** commands do not generate fictitious interfaces like eth0:1, eth0:2 and so on. This is a legacy naming scheme coming from the 2.0 kernel version and nowadays no more mandated. For example:

```
# ip addr add 192.168.0.11/24 dev eth0
# ip addr show eth0
2: eth0: <BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast qlen 100
    link/ether 00:48:54:1b:25:30 brd ff:ff:ff:ff:ff:ff
    inet 192.168.0.1/24 brd 192.168.0.255 scope global eth0
    inet 192.168.0.11/24 scope global secondary eth0
    inet6 fe80::248:54ff:fe1b:2530/10 scope link
# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:48:54:1B:25:30
          inet addr:192.168.0.1  Bcast:192.168.0.255  Mask:255.255.255.0
          inet6 addr: fe80::248:54ff:fe1b:2530/10 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:0 (0.0 b)  TX bytes:528 (528.0 b)
          Interrupt:9 Base address:0x5000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:34732 errors:0 dropped:0 overruns:0 frame:0
          TX packets:34732 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:3386912 (3.2 Mb)  TX bytes:3386912 (3.2 Mb)

foo       Link encap:IPv6-in-IPv4
          inet6 addr: 3ffe:9001:210:3::42/64 Scope:Global
          inet6 addr: fe80::c0a8:1/10 Scope:Link
          UP POINTOPOINT RUNNING NOARP  MTU:1480  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
```

```
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)
```

Our addictional IP address is reported by **ip addr show** and works, but **ifconfig** doesn't even know of its existence! To solve the problem we can use the *label* parameter:

```
# ip addr add 192.168.0.11/24 label eth0:1 dev eth0
# ip addr show eth0
2: eth0: <BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast qlen 100
    link/ether 00:48:54:1b:25:30 brd ff:ff:ff:ff:ff:ff
    inet 192.168.0.1/24 brd 192.168.0.255 scope global eth0
    inet 192.168.0.11/24 scope global secondary eth0:1
    inet6 fe80::248:54ff:fe1b:2530/10 scope link
# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:48:54:1B:25:30
          inet addr:192.168.0.1  Bcast:192.168.0.255  Mask:255.255.255.0
          inet6 addr: fe80::248:54ff:fe1b:2530/10 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:0 (0.0 b)  TX bytes:528 (528.0 b)
          Interrupt:9 Base address:0x5000

eth0:1    Link encap:Ethernet  HWaddr 00:48:54:1B:25:30
          inet addr:192.168.0.11  Bcast:0.0.0.0  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          Interrupt:9 Base address:0x5000
```

Notice that we can choose any arbitrary string as the label. We're not forced to use the 2.0 naming scheme; we must comply to it only if we care having backward compatibility with **ifconfig**.

5.3. Which IP for the tunnel

Choosing a global/public IP address (respectively an IPv6 address for SIT/IPv6-in-IPv4 tunnels and an IPv4 address for GRE/IPv4-in-IPv4 tunnels) for the local endpoint of the tunnel is probably the best thing we can do when our computer is a single host and not a router providing IPv6 connectivity to a whole LAN.

Instead, if we're configuring a router, we'd better use a link-local address for SIT/IPv6-in-IPv4 tunnels (in IPv6 link-local addresses are assigned automatically by means of stateless address autoconfiguration or manually configured) and a private address for GRE/IPv4-in-IPv4 tunnels (IPv4 has no link-local addresses). The valid address will then be only on eth0 (or the interface on the LAN side). Notice that in this configuration you need to activate forwarding among interfaces, using these commands:

```
sysctl -w net.ipv4.conf.all.forwarding=1  # for GRE (IPv4-in-IPv4)
```

```
sysctl -w net.ipv6.conf.all.forwarding=1  # for SIT (IPv6-in-IPv4)
```

For IPv4 you can even decide to enable forwarding only between a couple of interfaces, in this case you could use these commands:

```
sysctl -w net.ipv4.conf.eth0.forwarding=1
sysctl -w net.ipv4.conf.pippo.forwarding=1
```

Warning

meaning of this switch is different for IPv6 and doesn't work as expected, see kernel documentation for more information.

6. Routing

Now that our tunnel is configured, we have to specify which traffic will be directed through it. For IPv6 the most common choice is the following:

```
ip route add 2000::/3 dev foo
```

This way all IPv6 traffic going to addresses starting with 3 bits equal to 001 (that is, all global unicast IPv6 address space) will be directed to the foo interface. This is only one 8th of the available IPv6 address space, but you are guaranteed that every possible remote host will be in this range.

We can see the IPv4 routing table this way:

```
# ip route
192.168.0.0/24 dev eth0  scope link
127.0.0.0/8 dev lo  scope link
```

and the IPv6 routing table this way:

```
# ip -6 route
2000::/3 dev foo  proto kernel  metric 256  mtu 1480 advmss 1420
fe80::/10 dev eth0  proto kernel  metric 256  mtu 1500 advmss 1440
fe80::/10 dev foo  proto kernel  metric 256  mtu 1480 advmss 1420
ff00::/8 dev eth0  proto kernel  metric 256  mtu 1500 advmss 1440
ff00::/8 dev foo  proto kernel  metric 256  mtu 1480 advmss 1420
default dev eth0  proto kernel  metric 256  mtu 1500 advmss 1440
unreachable default dev lo  metric -1  error -101
```

If you need to specify a gateway (this is not for tunnels) then you can add the *via* parameter, for example:

```
ip route add 192.168.1.0/24 via 192.168.0.254 dev eth0
```

To remove a route you can obviously use **ip route del** but be careful: if you write **ip route del default** you're removing the default IPv4 route, not the IPv6 one! To remove the IPv6 default destination you need to use **ip -6 route del default**.

7. Practical applications

7.1. A complete example

This is a typical IPv6 tunnel for 6bone:

```
ip tunnel add $TUNNEL mode sit local any remote $V4_REMOTEADDR ttl 64
ip link   set $TUNNEL up
ip addr   add $V6_LOCALADDR dev $TUNNEL
ip route  add 2000::/3      dev $TUNNEL
```

where *$TUNNEL* is an arbitrary name assigned to the tunnel, *$V4_REMOTEADDR* is the IPv4 address of the remote end of the tunnel and *$V6_LOCALADDR* is the IPv6 local address assigned to our host. We've used the `any` value for the *local* endpoint address because this way we can handle a dynamic IPv4 address (e.g. assigned by a dialup connection to the ISP). Obviously we need to inform our tunnel broker when our address changes but this is out of the scope of this writing, also because there's no general standard procedure.

To shut down the tunnel:

```
ip tunnel del $TUNNEL
```

also automatically removes the routing entry and the address.

7.2. Comfort

Now, after we made sure everything works, we can use previous commands in a script called `ip-up.local` and saved in `/etc/ppp/`. This way, those commands will be automatically executed *everytime we connect PPP*. If we wanted to also automatically delete the tunnel upon PPP disconnection, we can create another script in the same directory, and call it `ip-down.local`.

As an example, if our tunnel broker is NGNET, we could use this script as `ip-up.local`:

```
#!/usr/bin/perl
##########################################################################
# Auto-setup script for NGNET's Tunnel Broker.
##########################################################################

# Configuration, fill with your values
# ------------------------------------
my $username  = '';
my $password  = '';
my $interface = '';
my $v6hname   = '';

# Don't touch anything below this line
# ------------------------------------

my $ngnet_tb  = '163.162.170.173';
my $ipv6_pref = '2001:06b8:0000:0400::/64';
my $url       = 'https://tb.ngnet.it/cgi-bin/tb.pl';

use strict;
use IO::File;
use LWP::UserAgent;

# Get our IPv4 address
my $lines;
my $f = IO::File->new();
$f->open("/sbin/ip addr show dev $interface|") or die("$!\n");
$f->read($lines, 4096);
$f->close();
$lines =~ /(\d+\.\d+\.\d+\.\d+)/ or die('Impossible condition');
my $v4addr = $1;

# Logging in
my $ua = LWP::UserAgent->new(keep_alive => 5);
my $resp = $ua->post($url, {
 oper     => 'reg_accesso',
 username => $username,
 password => $password,
 submit   => 'Submit'
});
$resp->is_success() or die('Failed reg_accesso: '.$resp->message);
$resp->as_string =~ /name=sid.*value=\"([^\"]+)\"/i or die('Missing sid');
my $sid = $1;

# Retrieve IPv6 addresses
my $myipv6;
my $ipv6end;
$resp = $ua->post($url, {
 oper        => 'tunnel_info',
 sid         => $sid,
 username    => $username,
 submit      => 'Submit'
});
```

```
$resp->is_success() or die('Failed tunnel_info: '.$resp->message);
$resp->as_string =~ /name=ipv6client.*value=\"([^\"]+)\"/i and $myipv6 = $1;
$resp->as_string =~ /name=ipv6server.*value=\"([^\"]+)\"/i and $ipv6end = $1;
die("missing IPv6 endpoints") unless ($myipv6 and $ipv6end);

# Extend tunnel lifetime
$resp = $ua->post($url, {
 oper        => 'tunnel_extend',
 sid         => $sid,
 username    => $username,
 submit      => 'Submit'
});
$resp->is_success() or die('Failed tunnel_extend: '.$resp->message);

# Update parameters on the remote side
$resp = $ua->post($url, {
 oper        => 'update_parameter',
 sid         => $sid,
 os_type     => 'Linux',
 ipv4client  => $v4addr,
 fl_entry    => $v6hname,
 username    => $username,
 ipv6_pref   => $ipv6_pref,
 submit      => 'Submit'
});
$resp->is_success() or die('Failed update_parameter: '.$resp->message);

# Set up tunnel on our side
system("/sbin/modprobe ipv6");
system("/sbin/ip tunnel add ngnet mode sit local any remote $ngnet_tb ttl 64");
system("/sbin/ip link set ngnet up");
system("/sbin/ip addr add $myipv6 dev ngnet");
system("/sbin/ip route add 2000::/3 dev ngnet");
```

`ip-down.local` could be:

```
#!/bin/bash
/sbin/ip tunnel del ngnet
```

8. Thanks

Thank to Giacomo Piva for pppd and NGNET integration idea.

References

Here are some useful links:

[IANA] Internet assigned numbers authority (http://www.iana.org/).

[ftpsite] *iproute2 ftp site (ftp://ftp.inr.ac.ru/ip-routing/)*.

[RFC2784] *Generic Routing Encapsulation (GRE) (http://www.ietf.org/rfc/rfc2784.txt)*, IETF, March 2000, Farinacci, Li, Hanks, Meyer, Traina.

[RFC2373] *IP Version 6 Addressing Architecture (http://www.ietf.org/rfc/rfc2373.txt)*, IETF, July 1998, Hinden, Deering.

[RFC2893] *Transition Mechanisms for IPv6 Hosts and Routers (http://www.ietf.org/rfc/rfc2893.txt)*, IETF, August 2000, Gilligan, Nordmark.

[NGNET] Telecom Italia Lab NGNET (http://www.ngnet.it/).