D.2. ip route

Another part of the **iproute2** suite of tools for IP management, **ip route** provides management tools for manipulating any of the routing tables. Operations include displaying routes or the routing cache, adding routes, deleting routes, modifying existing routes, and fetching a route and clearing an entire routing table or the routing cache.

One thing to keep in mind when using the **ip route** is that you can operate on any of the 255 routing tables with this command. Where the **route** command operated only on the main routing table (table 254), the **ip route** command operates by default on the main routing table, but can be easily coaxed into using other tables with the `table` parameter.

Fortunately, as mentioned earlier, the **iproute2** suite of tools does not rely on DNS for any operation so, the ubiquitous `-n` switch in previous examples will not be required in any example here.

All operations with the **ip route** command are atomic, so each command will return either `RTNETLINK answers: No such process` in the case of an error, or nothing in the face of success. The `-s` switch which provides additional statistical information when reporting link layer information will only provide additional information when reporting on the state of the routing cache or fetching a specific route..

The **ip route** utility when used in conjunction with the **ip rule** utility can create stateless NAT tables. It can even manipulate the local routing table, a routing table used for traffic bound for broadcast addresses and IP addresses hosted on the machine itself.

In order to understand the context in which this tool runs, you need to understand some of the basics of IP routing, so if you have read the above introduction to the **ip route** tool, and are confused, you may want to read Chapter 4, *IP Routing* and grasp some of the concepts of IP routing (with linux) before continuing here.

D.2.1. Displaying a routing table with ip route show

In its simplest form, **ip route** can be used to display the main routing table output. The output of this command is significantly different from the output of the **route**. For comparison, let's look at the output of both **route -n** and **ip route show**.

Example D.11. Viewing the main routing table with ip route show

```
[root@tristan]# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
192.168.99.0    0.0.0.0         255.255.255.0   U     0      0        0 eth0
127.0.0.0       0.0.0.0         255.0.0.0       U     0      0        0 lo
0.0.0.0         192.168.99.254  0.0.0.0         UG    0      0        0 eth0
[root@tristan]# ip route show
192.168.99.0/24 dev eth0  scope link
127.0.0.0/8 dev lo  scope link
default via 192.168.99.254 dev eth0
```

If you are accustomed to the **route** output format, the **ip route** output can seem terse. The same basic information is displayed, however. As with our former example, let's ignore the 127.0.0.0/8 loopback route for the moment. This is a required route for any IPs hosted on the loopback interface. We are far more interested in the other two routes.

The network 192.168.99.0/24 is available on eth0 with a scope of link, which means that the network is valid and reachable through this device (eth0). Refer to Table C.2, "IP Scope under ip address" for definitions of possible scopes. As long as link remains good on that device, we should be able to reach any IP address inside of 192.168.99.0/24 through the eth0 interface.

Finally, our all-important default route is expressed in the routing table with the word default. Note that any destination which is reachable through a gateway appears in the routing table output with the keyword `via`. This final line matches semantically with the final line of output from **route -n** above.

Now, let's have a look at the local routing table, which we can't see with **route**. To be fair, it is usually completely unnecessary to view and/or manipulate the local routing table, which is why **route** provides no way to access this information.

Example D.12. Viewing the local routing table with ip route show table local

```
[root@tristan]# ip route show table local
local 192.168.99.35 dev eth0  proto kernel  scope host  src 192.168.99.35
broadcast 127.255.255.255 dev lo  proto kernel  scope link  src 127.0.0.1
broadcast 192.168.99.255 dev eth0  proto kernel  scope link  src 192.168.99.35
broadcast 127.0.0.0 dev lo  proto kernel  scope link  src 127.0.0.1
local 127.0.0.1 dev lo  proto kernel  scope host  src 127.0.0.1
local 127.0.0.0/8 dev lo  proto kernel  scope host  src 127.0.0.1
```

This gives us a good deal of information about the IP networks to which the machine is directly connected, and an inside look into the way that the routing tables treat special addresses like broadcast addresses and locally configured addresses.

The first field in this output tells us whether the route is for a broadcast address or an IP address or range locally hosted on this machine. Subsequent fields inform us through which device the destination is reachable, and notably (in this table) that the kernel has added these routes as part of bringing up the IP layer interfaces.

For each IP hosted on the machine, it makes sense that the machine should restrict accessiblity to that IP or IP range to itself only. This explains why, in Example D.12, “Viewing the local routing table with ip route show table local”, 192.168.99.35 has a host scope. Because `tristan` hosts this IP, there's no reason for the packet to be routed off the box. Similarly, a destination of localhost (127.0.0.1) does not need to be forwarded off this machine. In each of these cases, the scope has been set to host.

For broadcast addresses, which are intended for any listeners who happen to share the IP network, the destination only makes sense as for a scope of devices connected to the same link layer [49].

The final characteristic available to us in each line of the local routing table output is the `src` keyword. This is treated as a hint to the kernel about what IP address to select for a source address on outgoing packets on this interface. Naturally, this is most commonly used (and abused) on multi-homed hosts, although almost every machine out there uses this hint for connections to localhost [50].

Now that we have inspected the main routing table and the local routing table, let's see how easy it is to look at any one of the other routing tables. This is as simple as specifying the table by its name in `/etc/iproute2/rt_tables` or by number. There are a few reserved table identifiers in this file, but the other table numbers between 1 and 252 are available for the user. Please note that this example is for demonstration only and has no intrinsic value other than showing the use of the `table` parameter.

Example D.13. Viewing a routing table with ip route show table

```
[root@tristan]# ip route show table special
Error: argument "special" is wrong: table id value is invalid

[root@tristan]# echo 7 special >> /etc/iproute2/rt_tables
[root@tristan]# ip route show table special
[root@tristan]# ip route add table special default via 192.168.99.254
[root@tristan]# ip route show table special
default via 192.168.99.254 dev eth0
```

In the above example you get a first glance at how to add a route to a table other than the main routing table, but what we are really interested in is the final command and output. In Example D.13, “Viewing a routing table with ip route show table”, we have identified table 7 by the name "special" and have added a

route to this table. The command `ip route show table special` shows us routing table number 7 from the kernel.

ip route consults `/etc/iproute2/rt_tables` for a table identifier. If it finds no identifier, it complains that it cannot find a reference to such a table. If a table identifier is found, then the corresponding routing table is displayed.

The use of multiple routing tables can make a router very complex, very quickly. Using names instead of numbers for these tables can assist in the management of this complexity. For further discussion on managing multiple routing tables and some issues of handling them see Section 10.3, “Using the Routing Policy Database and Multiple Routing Tables” .

D.2.2. Displaying the routing cache with ip route show cache

The routing cache is used by the kernel as a lookup table analogous to a quick reference card. It's faster for the kernel to refer to the cache (internally implemented as a hash table) for a recently used route than to lookup the destination address again. Routes existing in the route cache are periodically expired.

The routing cache can be displayed in all its glory with **ip route show cache**, which provides a detailed view of recent destination IP addresses and salient characteristics about those destinations. On routers, masquerading boxen and firewalls, the routing cache can become very large. Instead of viewing the entire routing cache even on a workstation, we'll select a particular destination from the routing cache to examine.

Example D.14. Displaying the routing cache with ip route show cache

```
[root@tristan]# ip route show cache 192.168.100.17
192.168.100.17 from 192.168.99.35 via 192.168.99.254 dev eth0
    cache  mtu 1500 rtt 18ms rttvar 15ms cwnd 15 advmss 1460
192.168.100.17 via 192.168.99.254 dev eth0  src 192.168.99.35
    cache  mtu 1500 advmss 1460
```

FIXME! I don't know how to explain rtt, rttvar, and cwnd, even after reading Alexey's comments in the iproute2 documentation! Not only that, I'm not sure why there are two entries!

The output in Example D.14, “Displaying the routing cache with ip route show cache” summarizes the reachability of the destination 192.168.100.17 from 192.168.99.35. The first line of each entry provides some important information for us: the destination IP, the source IP, the gateway through which the destination is reachable, and the interface through which packets were routed. Together, these data identify a route entry in the cache.

Characteristics of that route are summarized in the second line of each entry. For the route between `tristan` and `isolde`, we see that Path MTU discovery has identified 1500 bytes as the maximum packet size from end to end. The maximum segment size (MSS) of data is 1460 bytes. Although this is not usually of any but the most casual of interest, it can be helpful diagnostic information.

If you are a die-hard fan of statistics, and can't get enough information about the routing on your machine, you can always throw the `-s` switch.

Example D.15. Displaying statistics from the routing cache with ip -s route show cache

```
[root@tristan]# ip -s route show cache 192.168.100.17
192.168.100.17 from 192.168.99.35 via 192.168.99.254 dev eth0
    cache  users 1 used 326 age 12sec mtu 1500 rtt 72ms rttvar 22ms cwnd 2 advmss 1460
192.168.100.17 via 192.168.99.254 dev eth0  src 192.168.99.35
    cache  users 1 used 326 age 12sec mtu 1500 advmss 1460
```

With this output, you'll get just a bit more information about the routes. The most interesting datum is usually the "used" field, which indicates the number of times this route has been accessed in the routing cache. This can give you a very good idea of how many times a particular route has been used. The age field

is used by the kernel to decide when to expire a cache entry. The age is reset every time the route is accessed [51].

In sum, you can use the routing cache to learn a good deal about remote IP destinations and some of the characteristics of the network path to those destinations.

D.2.3. Using ip route add to populate a routing table

ip route add is a used to populate a routing table. Although you can use **route add** to do the same thing, **ip route add** offers a large number of options that are not possible with the venerable **route** command. After we have looked at some simple examples, we'll discuss more complex routes with **ip route**.

In Section D.1, "**route**", we used two classic examples of adding a network route (to our service provider's network from) and a host route. Let's look at the difference in syntax with the **ip route** command.

Example D.16. Adding a static route to a network with route add, cf. Example D.4, "Adding a static route to a network route add"

```
[root@masq-gw]# ip route add 10.38.0.0/16 via 192.168.100.1
```

This is one of the simplest examples of the syntax of the **ip route**. As you may recall, you can only add a route to a destination network through a gateway that is itself already reachable. In this case, `masq-gw` already knows a route to 192.168.100.1 (`service-router`). Now any packets bound for 10.38.0.0/16 will be forwarded to 192.168.100.1.

Other interesting examples of this command involve the use of `prohibit` and `from`. Use of the `prohibit` will cause the router to report that the requested destination is unreachable. If you know a netblock that hosts a service you are not interested in allowing your users to access, this is an effective way to block the outbound connection attempts.

Let's look at an example of **tcpdump** output which shows the `prohibit` route in action.

Example D.17. Adding a `prohibit` route with route add

```
[root@masq-gw]# ip route add prohibit 209.10.26.51
[root@tristan]# ssh 209.10.26.51
ssh: connect to address 209.10.26.51 port 22: No route to host
[root@masq-gw]# tcpdump -nnq -i eth2
tcpdump: listening on eth2
22:13:13.740406 192.168.99.35.51973 > 209.10.26.51.22: tcp 0 (DF)
22:13:13.740714 192.168.99.254 > 192.168.99.35: icmp: host 209.10.26.51 unreachable - admin prohibited filter [tos 0xc0]
```

Compare the ICMP packet returned to the sender in this case with the ICMP packet returned if you used **iptables** and the `REJECT` target [52]. Although the net effect is identical (the user is unable to reach the intended destinatioan), the user gets two different error messages. With an **iptables** `REJECT`, the user sees `Connection refused`, where the user sees `No route to host` with the use of `prohibit`. These are but two of the options for controlling outbound access from your network.

Supposing you don't want to block access to this particular host for all of your users, the `from` option comes to your aid.

Example D.18. Using `from` in a routing command with route add

```
[root@masq-gw]# ip route add prohibit 209.10.26.51 from 192.168.99.35
```

Now, you have effectively blocked the source IP 192.168.99.35 from reaching 209.10.26.51. Any packets matching this source and destination address will match this route. In this case, `masq-gw` will generate an ICMP error message indicating that the destination is administratively unreachable.

If you are still following along here, you can see that the options for identifying particular routes are many and multi-faceted. The `src` option provides a hint to the kernel for source address selection. When you are working with multiple routing tables and different classes of traffic, you can ease your administrative burden, by hosting several different IPs on your linux machine and setting the source address differently, depending on the type of traffic.

In the example below, let's assume that our masquerading host also runs a DNS resolver for the internal network and we have selected all of the outbound DNS packets to be routed according to table 7 [53]. Now, any packet which originates on this box (or is masqueraded through this table) will have its source IP set to 205.254.211.198.

Example D.19. Using `src` in a routing command with route add

```
[root@masq-gw]# ip route add default via 205.254.211.254 src 205.254.211.198 table 7
```

FIXME!! I have nothing to say about `nexthop` yet, because I have never used it, this goes for `equalize` and `onlink` as well. If anybody has some examples s/he would like to contribute, I'd love to hear.

There are other options to the **ip route add** documented in Alexey's thorough **iproute2** documentation. For further research, I'd suggested acquiring and reading this manual.

D.2.4. Adding a default route with ip route add default

Naturally, one of the most important routes on a machine is its default route. Adding a default route is one of the simplest operations with **ip route**.

We need exactly one piece of information in order to set the default route on a machine. This is the IP address of the gateway. The syntax of the command is extremely simple and aside from the use of the `via` instead of `gw`, it is almost the same command as the equivalent **route -n**.

Example D.20. Setting the default route with ip route add default

```
[root@tristan]# ip route add default via 192.168.99.254
```

D.2.5. Setting up NAT with ip route add nat

Be sure to see Chapter 5, *Network Address Translation (NAT)* for a full treatment of the issues involved in network address translation (NAT). If you are here to learn a bit more about how to set up NAT in your network, then you should know that the **ip route add nat** is only half of the solution. You must understand that performing NAT with **iproute2** involves one component to rewrite the inbound packet (**ip route add nat**), and another command to rewrite the outbound packet (**ip rule add nat**). If you only get half of the system in place, your NAT will only work halfway--or not at all, depending on how you define "work".

Alexey documents clearly in the appendix to the **iproute2** manual that the NAT provided by the **iproute2** suite is stateless. This is distinctly unlike NAT with netfilter. Refer to Section 5.5, "Destination NAT with netfilter (DNAT)" and Section 8.3, "Netfilter Connection Tracking" for a better look at the connection tracking and network address translation support available under netfilter.

The **ip route add nat** command is used to rewrite the destination address of a packet from one IP or range to another IP or range. The **iproute2** tools can only operate on the entire IP packet. There is no provision directly within the **iproute2** suite to support conditional rewriting based on the destination port of a UDP datagram or TCP segment. It's the whole packet, every packet, and nothing but the packet [54].

Example D.21. Creating a NAT route for a single IP with ip route add nat

```
[root@masq-gw]# ip route add nat 205.254.211.17 via 192.168.100.17
[root@masq-gw]# ip route show table local | grep ^nat
nat 205.254.211.17 via 192.168.100.17  scope host
```

The route entry we have just made tells the kernel to rewrite any inbound packet bound for 205.254.211.17 to 192.168.100.17. The actual rewriting of the packet occurs at the routing stage of the packets trip through the kernel. This is an important detail, illuminated more fully in Section 5.4, "Stateless NAT and Packet Filtering".

Not only can **iproute2** support network address translation for single IPs, but also for entire network ranges. The syntax is substantially similar to the syntax above, but uses a CIDR network address instead of a single IP.

Example D.22. Creating a NAT route for an entire network with ip route add nat

```
[root@masq-gw]# ip route add nat 205.254.211.32/29 via 192.168.100.32
[root@masq-gw]# ip route show table local | grep ^nat
nat 205.254.211.32/29 via 192.168.100.32  scope host
```

In this example, we are adding a route for an entire network. Any IP packets which come to us destined for any address between 205.254.211.32 and 205.254.211.39 will be rewritten to the corresponding address in the range 192.168.100.32 through 192.168.100.39. This is a shorthand way to specify multiple translations with CIDR notation.

Again, this is only one half of the story for NAT with **iproute2**. Please be certain to read the section below for usage information on **ip rule add nat**, in addition to Chapter 5, *Network Address Translation (NAT)* which will provide fuller documentation for NAT support under linux. Don't forget to use **ip route flush cache** after you add NAT routes and the corresponding NAT rules [55].

D.2.6. Removing routes with ip route del

The **ip route del** takes exactly the same syntax as the **ip route add** command, so if you have familiarized yourself with the syntax, this should be a snap.

It is, in fact, almost trivial to delete routes on the command line with **ip route del**. You can simply identify the route you wish to remove with **ip route show** command and append the output line verbatim to **ip route del**.

Example D.23. Removing routes with ip route del [56]

```
[root@masq-gw]# ip route show
192.168.100.0/30 dev eth3  scope link
205.254.211.0/24 dev eth1  scope link
192.168.100.0/24 dev eth0  scope link
192.168.99.0/24 dev eth0  scope link
192.168.98.0/24 via 192.168.99.1 dev eth0
10.38.0.0/16 via 192.168.100.1 dev eth3
127.0.0.0/8 dev lo  scope link
default via 205.254.211.254 dev eth1
[root@masq-gw]# ip route del 10.38.0.0/16 via 192.168.100.1 dev eth3
```

We identified the network route to 10.38.0.0/16 as the route we wished to remove, and simply appended the description of the route to our **ip route del** command.

This command can be used to remove routes such as broadcast routes and routes to locally hosted IPs in addition to manipulation of any of the other routing tables. This means that you can cause some very strange problems on your machine by inadvertently removing routes, especially routes to locally hosted IP addresses.

D.2.7. Altering existing routes with ip route change

Occasionally, you'll want to remove a route and replace it with another one. Fortunately, this can be done atomically with **ip route change**.

Let's change the default route on tristan with this command.

Example D.24. Altering existing routes with ip route change

```
[root@tristan]# ip route change default via 192.168.99.113 dev eth0
[root@tristan]# ip route show
192.168.99.0/24 dev eth0  scope link
127.0.0.0/8 dev lo  scope link
default via 192.168.99.113 dev eth0
```

If you do use the **ip route change** command, you should be aware that it does not communicate a routing table state change to the routing cache, so here is another good place to get in the habit of using **ip route flush cache**.

There's not much more to say about the use of this command. If you don't want to use an **ip route del** immediately followed by an **ip route add** you can use **ip route change**.

D.2.8. Programmatically fetching route information with ip route get

When configuring routing tables, it is not always sufficient to search for the destination manually. Especially with large routing tables, this can become a rather boring and time-consuming endeavor. Fortunately, **ip route get** elegantly solves the problem. By simulating a request for the specified destination, **ip route get** causes the routing selection algorithm to be run. When this is complete, it prints out the resulting path to the destination. In one sense, this is almost equivalent to sending an ICMP echo request packet and then using **ip route show cache**.

Example D.25. Testing routing tables with ip route get

```
[root@tristan]# ip -s route get 127.0.0.1/32
ip -s route get 127.0.0.1/32
local 127.0.0.1 dev lo  src 127.0.0.1
    cache <local>  users 1 used 1 mtu 16436 advmss 16396
[root@tristan]# ip -s route get 127.0.0.1/32
local 127.0.0.1 dev lo  src 127.0.0.1
    cache <local>  users 1 used 2 mtu 16436 advmss 16396
```

For casual use, **ip route get** is an invaluable tool. An obvious side effect of using **ip route get** the increase in the usage count of every touched entry in the routing cache. While this is no problem, it will alter the count of packets which have used that particular route. If you are using **ip** to count outbound packets (people have done it!) you should be cautious with this command.

D.2.9. Clearing routing tables with ip route flush

The `flush` option, when used with **ip route** empties a routing table or removes the route for a particular destination. In Example D.26, “Removing a specific route and emptying a routing table with ip route flush”, we'll first remove a route for a destination network using **ip route flush**, and then we'll remove all of the routes in the main routing table with one command.

If you do not wish to delete routes by hand, you can quickly empty all of the routes in a table by specifying a table identifier to the **ip route flush** command.

Example D.26. Removing a specific route and emptying a routing table with ip route flush

```
[root@masq-gw]# ip route flush
"ip route flush" requires arguments
[root@masq-gw]# ip route flush 10.38
Nothing to flush.
[root@masq-gw]# ip route flush 10.38.0.0/16
[root@masq-gw]# ip route show
192.168.100.0/30 dev eth3  scope link
205.254.211.0/24 dev eth1  scope link
192.168.100.0/24 dev eth0  scope link
192.168.99.0/24 dev eth0  scope link
192.168.98.0/24 via 192.168.99.1 dev eth0
```

```
127.0.0.0/8 dev lo  scope link
default via 205.254.211.254 dev eth1
[root@masq-gw]# ip route flush table main
[root@masq-gw]# ip route show
[root@masq-gw]#
```

Note that you should exercise caution when using **ip route flush table** because you can easily destroy your own route to the machine by specifying the main routing table or a routing table that is used to send packets to your workstation. Naturally, this is not a problem if you are connected to the machine via a serial, modem, console, or other out of band connection.

D.2.10. ip route flush cache

Above, in Section D.2.2, "Displaying the routing cache with **ip route show cache**", we looked at the contents of the routing cache, a hash table in the kernel which contains recently used routes. To quote John S. Denker, you should not forget to use **ip route flush cache** after you have changed the routing tables; "otherwise changes will take effect only after some maddeningly irreproducible delay." [57]

Since the kernel refers to the routing cache before fetching a new route from the routing tables, **ip route flush cache** empties the cache of any data. Now when the kernel goes to the routing cache to locate the best route to a destination, it finds the cache empty. Next, it traverses the routing policy database and routing tables. When the kernel finds the route, it will enter the newly fetched destination into the routing cache.

Example D.27. Emptying the routing cache with ip route flush cache

```
[root@tristan]# ip route show cache
local 127.0.0.1 from 127.0.0.1 tos 0x10 dev lo
    cache <local>  mtu 16436 advmss 16396
local 127.0.0.1 from 127.0.0.1 dev lo
    cache <local>  mtu 16436 advmss 16396
192.168.100.17 from 192.168.99.35 via 192.168.99.254 dev eth0
    cache  mtu 1500 rtt 18ms rttvar 15ms cwnd 15 advmss 1460
192.168.100.17 via 192.168.99.254 dev eth0  src 192.168.99.35
    cache  mtu 1500 advmss 1460
[root@tristan]# ip route flush cache
[root@tristan]# ip route show cache
[root@tristan]# ip route show cache
local 127.0.0.1 from 127.0.0.1 tos 0x10 dev lo
    cache <local>  mtu 16436 advmss 16396
local 127.0.0.1 from 127.0.0.1 dev lo
    cache <local>  mtu 16436 advmss 16396
```

When making routing changes to a linux box, you can save yourself some troubleshooting time (and confusion) by getting in the habit of finishing your routing commands with **ip route flush cache**.

D.2.11. Summary of the use of ip route

With this overview of the use of the **ip route** utility, you should be ready to step into some advanced territory to harness multiple routing tables, take advantage of special types of routes, use network address translation, and gather detailed statistics on the usage of your routing tables.

---

[49] I'm going to specifically neglect a discussion of bridging and broadcast addresses for now. Let's assume a simple Ethernet where the entire IP network is on one hub or switch.

[50] When a user initiates a connection to localhost (let's say localhost:25, where a private SMTP server is listening), the connection could, of course, come from the IP assigned to any of the Ethernet interfaces. It makes the most sense, however, for the source IP to be set to 127.0.0.1, since the connection is actually initiated from on the local machine. Some services running on a local machine rely on the loopback interface and will restrict incoming connections to source addresses of 127.0.0.1. Frankly, I find this quite sensible for services which are not intended for public use.

[51] Be wary of using **ip route get** and **ip route show cache** because **ip route get** implicitly causes a route lookup to be performed, thus increasing the used counter on the route, and resetting the age. This will alter the statistics reported by **ip -s route show cache**.

[52] Please note that I in the cross-referenced example I have used **iptables**. The same behaviour should be expected with **ipchains**. (Anybody have any proof?)

[53] If you wonder how this kind of magic is accomplished, you'll want to read Section 10.3.2, "Using fwmark for Policy Routing" .

[54] This should not lead you into believing it cannot be done. This is linux after all! By routing via fwmark, and using the `--mark` option to **ipchains** or the MARK target and `--set-mark` option in **iptables**, you can perform conditional routing based on characteristics and contents of the packet.

[55] You can always use my SysV initialization script and configuration file instead of entering your own commands, however, it is always important to understand the tool you are using.

[56] Please note that this is the same routing table as is shown in the Example D.2, "Viewing a complex routing table with route" , which displays the output from **route -n** on `masq-gw`.

[57] See this remark in his documentation of a workaround with FreeS/WAN and iproute2 to approximate more RFC-like SPD behaviour for a linux IPSec tunnel.

Prev D.1. **route** | Up Home | Next D.3. **ip rule**