Overview

iproute2 is the Linux networking toolkit that replaced net-tools (ifconfig, route, arp etc.)

Old style network utilities like ifconfig and route are still there just for backwards compatibility and do not provide access to new features like policy-based routing or network namespaces.

Note that iproute2 has been a **standard Linux tool** since the early 2000's. It's included in every distro by default, or at least available from the repos (OpenWRT is one of the cases).

iproute2 was originally written by Alex Kuznetsov and is now maintained by Stephen Hemminger.

This document aims to provide comprehensive but easy to use documentation for the ip command included in iproute2 package. There are more, such as ss (netstat replacement, fairly straightforward), tc (QoS management), but documenting them in this style, especially tc, would be a separate big project.

Instead of listing commands and describing what they do, it lists common tasks network administrators need to perform and gives commands to solve them, hence cheatsheet.

Contributions are always welcome, you can find the "source code" at github.com/dmbaturin/iproute2-cheatsheet https://github.com/dmbaturin/iproute2-cheatsheet .

```
git clone https://github.com/dmbaturin/iproute2-cheatsheet.git
```

This document is provided "as is", without any warranty. The authors are not liable for any damage related to using it.

General notes

All commands that change any settings (that is, not just display them) require root privileges.

There are configuration files in /etc/iproute2, mainly for assinging symbolic names to network stack entities such as routing tables. Those files are re-read every time you run the ip command and you don't need to do anything to apply the changes.

Typographic conventions

Metasyntactic variables are written in shell-style syntax, ${something}. Optional command parts are in square brackets.

Table of Contents:

Address management

In this section ${address} value should be a host address in dotted decimal format, and ${mask} can be either a dotted decimal subnet mask or a prefix length. That is, both 192.0.2.10/24 and 192.0.2.10/255.255.255.0 are equally acceptable.

If you are not sure if something is a correct host address, use ipcalc or similar program to check.

Show all addresses

```
ip address show
```

All show commands can be used with -4 or -6 options to show only IPv4 or IPv6 addresses.

Show addresses for a single interface

```
ip address show ${interface name}
```

Examples:

```
ip address show eth0
```

Show addresses only for running interfaces

```
ip address show up
```

Show only static or dynamic IPv6 addresses

Show only statically configured addresses:

```
ip address show [dev ${interface}] permanent
```

Show only addresses learnt via autoconfiguration:

```
ip address show [dev ${interface}] dynamic
```

Add an address to an interface

```
ip address add ${address}/${mask} dev ${interface name}
```

Examples:

```
ip address add 192.0.2.10/27 dev eth0
```

```
ip address add 2001:db8:1::/48 dev tun10
```

You can add as many addresses as you want.

If you add more than one address, your machine will accept packets for all of them. The first address you added will be used as source address for outgoing traffic by default, it's often referred to as primary address.

Add an address with human-readable description

Add an address with human readable description

```
ip address add ${address}/${mask} dev ${interface name} label ${interface
name}:${description}
```

Examples:

```
ip address add 192.0.2.1/24 dev eth0 label eth0:WANaddress
```

A label must start with the interface name followed by a colon due to some backwards compatibility issues, otherwise you'll get an error. Keep the label shorter than sixteen characters, or else you'll get this error:

```
RTNETLINK answers: Numerical result out of range
```

Notes

For IPv6 addresses this command has no effect (address will be added, but without a label).

Delete an address

```
ip address delete ${address}/${prefix} dev ${interface name}
```

Examples:

```
ip address delete 192.0.2.1/24 dev eth0
```

```
ip address delete 2001:db8::1/64 dev tun1
```

Interface name is required. Linux does allow the same address to be configured on multiple interfaces and it has valid use cases.

Remove all addresses from an interface

```
ip address flush dev ${interface name}
```

Examples:

```
ip address flush dev eth1
```

By default this command removes both IPv4 and IPv6 addresses. If you want to remove only IPv4 or IPv6 addresses, use “ip -4 address flush” or “ip -6 address flush”.

Notes

Note that there is no way to rearrange addresses and replace the primary address. Make sure you set the primary address first.

Route management

For IPv4 routes, you can use either a prefix length or a dotted decimal subnet mask. That is, both 192.0.2.0/24 and 192.0.2.0/255.255.255.0 are equally acceptable.

Note: as per the section below, if you set up a static route, and it becomes useless because the interface goes down, it will be removed and **never get back on its own**. You may not have noticed this behaviour because in many cases additional software (e.g. NetworkManager or rp-pppoe) takes care of restoring routes associated with interfaces.

If you are going to use your Linux machine as a router, consider installing a routing protocol suite such as Quagga http://quagga.net or BIRD http://bird.network.cz . They keep track of interface status and restore routes when a link goes up after going down. Of course they also allow you to use dynamic routing protocols such as OSPF and BGP.

Connected routes

Some routes appear in the system without explicit configuration (against your will).

Once you assign an address to an interface, the system calculates its network address and creates a route to it (this is why the subnet mask is required). These routes are called connected routes.

For example, if you assign 203.0.113.25/24 to eth0, a connected route to 203.0.113.0/24 network will be created and the system will know that hosts from that network can be reached directly.

When an interface goes down, connected routes associated with it are removed. This is used for inaccessible gateway detection so routes through gateways that went inaccessible are removed. Same mechanism prevents you from creating routes through inaccessible gateways.

View all routes

```
ip route
```

```
ip route show
```

You can use -4 and -6 options to view only IPv4 or IPv6 routes. If no options given, IPv4 routes are displayed. To view IPv6 routes, use:

```
ip -6 route
```

View routes to a network and all its subnets

```
ip route show to root ${address}/${mask}
```

For example, if you use 192.168.0.0/24 subnet in your network and it's broken into 192.168.0.0/25 and 192.168.0.128/25, you can see all those routes with:

```
ip route show to root 192.168.0.0/24
```

Note: the word "to" in this and other show commands is optional.

View routes to a network and all supernets

```
ip route show to match ${address}/${mask}
```

```
    ip route show to match ${address}/${mask}
```

If you want to view routes to 192.168.0.0/24 and all larger subnets, use:

```
    ip route show to match 192.168.0.0/24
```

As routers prefer more specific routes to less specific, this is often useful for debugging in situations when traffic to specific subnet is sent the wrong way because a route to it is missing but routes to larger subnets exist.

View routes to exact subnet

```
    ip route show to exact ${address}/${mask}
```

If you want to see the routes to 192.168.0.0/24, but not to, say 192.168.0.0/25 and 192.168.0.0/16, you can use:

```
    ip route show to exact 192.168.0.0/24
```

View only the route actually used by the kernel

```
    ip route get ${address}/${mask}
```

Example:

```
    ip route get 192.168.0.0/24
```

Note that in complex routing scenarios like multipath routing, the result may be "correct but not complete", as it always shows one route that will be used first. In most situations it's not a problem, but never forget to look at the corresponding "show" command output too.

View route cache (pre 3.6 kernels only)

```
    ip route show cached
```

Until the version 3.6, Linux used route caching. In older kernels, this command displays the contents of the route cache. It can be used with modifiers described above. In newer kernels it does nothing.

Add a route via gateway

```
    ip route add ${address}/${mask} via ${next hop}
```

Examples:

```
    ip route add 192.0.2.128/25 via 192.0.2.1

    ip route add 2001:db8:1::/48 via 2001:db8:1::1
```

Add a route via interface

```
    ip route add ${address}/${mask} dev ${interface name}
```

Example:

```
    ip route add 192.0.2.0/25 dev ppp0
```

Interface routes are commonly used with point-to-point interfaces like PPP tunnels where next hop address is not required.

Change or replace a route

You may use "change" command to change parameters of existing routes. "Replace" command can be used to add new route or modify existing one if it doesn't exist. Examples:

```
ip route change 192.168.2.0/24 via 10.0.0.1
```

```
ip route replace 192.0.2.1/27 dev tun0
```

Delete a route

```
ip route delete ${rest of the route statement}
```

Examples:

```
ip route delete 10.0.1.0/25 via 10.0.0.1
```

```
ip route delete default dev ppp0
```

Default route

There is a shortcut to add default route.

```
ip route add default via ${address}/${mask}
```

```
ip route add default dev ${interface name}
```

These are equivalent to:

```
ip route add 0.0.0.0/0 via ${address}/${mask}
```

```
ip route add 0.0.0.0/0 dev ${interface name}
```

With IPv6 routes it also works and is equivalent to ::/0

```
ip -6 route add default via 2001:db8::1
```

Blackhole routes

```
ip route add blackhole ${address}/${mask}
```

Examples:

```
ip route add blackhole 192.0.2.1/32
```

Traffic to destinations that match a blackhole route is silently discarded.

Blackhole routes have dual purpose. First one is straightforward, to discard traffic sent to unwanted destinations, e.g. known malicious hosts.

The second one is less obvious and uses the "longest match rule" as per RFC1812. In some cases you may want the router to think it has a route to a larger subnet, while you are not using it as a whole, e.g. when advertising the whole subnet via dynamic routing protocols. Large subnets are commonly broken into smaller parts, so if your subnet is 192.0.2.0/24, and you have assigned 192.0.2.1/25 and 192.0.2.129/25 to your interfaces, your system creates connected routes to the /25's, but not the whole /24, and routing daemons may not want to advertise /24 because you have no route to that exact

/24, and routing daemons may not want to advertise /24 because you have no route to that exact subnet. The solution is to setup a blackhole route to 192.0.2.0/24. Because routes to smaller subnets are preferred over larger subnets, it will not affect actual routing, but will convince routing daemons there's a route to the supernet.

Other special routes

```
ip route add unreachable ${address}/${mask}

ip route add prohibit ${address}/${mask}

ip route add throw ${address}/${mask}
```

These routes make the system discard packets and reply with an ICMP error message to the sender.

unreachable
    Sends ICMP "host unreachable".
prohibit
    Sends ICMP "administratively prohibited".
throw
    Sends "net unreachable".

Unlike blackhole routes, these can't be recommended for stopping unwanted traffic (e.g. DDoS) because they generate a reply packet for every discarded packet and thus create even greater traffic flow. They can be good for implementing internal access policies, but consider firewall for this purpose first.

"Throw" routes may be used for implementing policy-based routing, in non-default tables they stop current table lookup, but don't send ICMP error messages.

Routes with different metric

```
ip route add ${address}/${mask} via ${gateway} metric ${number}
```

Examples:

```
ip route add 192.168.2.0/24 via 10.0.1.1 metric 5

ip route add 192.168.2.0 dev ppp0 metric 10
```

If there are several routes to the same network with different metric value, the one with **the lowest** metric will be preferred.

Important part of this concept is that when an interface goes down, routes that would be rendered useless by this event disappear from the routing table (see the Connected Routes #Connected routes section), and the system will fall back to higher metric routes.

This feature is commonly used to implement backup connections to important destinations.

Multipath routing

```
ip route add ${addresss}/${mask} nexthop via ${gateway 1} weight ${number} nexthop via
${gateway 2} weight ${number}
```

Multipath routes make the system balance packets across several links according to the weight (higher weight is preferred, so gateway/interface with weight of 2 will get roughly two times more traffic than another one with weight of 1). You can have as many gateways as you want and mix gateway and interface routes, like:

```
ip route add default nexthop via 192.168.1.1 weight 1 nexthop dev ppp0 weight 10
```

Warning: the downside of this type of balancing is that packets are not guaranteed to be sent back through the same link they came in. This is called "asymmetric routing". For routers that simply forward packets and don't do any local traffic processing such as NAT, this is usually normal, and in some cases even unavoidable.

If your system does anything but forwarding packets between interfaces, this may cause problems with incoming connections and some measures should be taken to prevent it.

Link management

Link is another name for network interface. Commands from "ip link" family perform operations that are common for all interface types, like viewing link information or changing the MTU.

They also can create many types of interfaces, except for tunnel (IPIP, GRE etc.) and L2TPv3 pseudowires that have their own commands.

Note that interface name you set with "name ${name}" parameter of "ip link add" and "ip link set" commands may be arbitrary, and may even contain unicode characters. However, it's better to stick with ASCII because other programs may not handle unicode correctly.

Also note that other programs, such as iptables, may have their own link name format and length restrictions, so it's better to use short alphanumeric names, and provide additional information in link aliases #Set human-readable link description .

Show information about all links

```
ip link show

ip link list
```

These commands are equivalent and can be used with the same arguments.

Show information about specific link

```
ip link show dev ${interface name}
```

Examples:

```
ip link show dev eth0

ip link show dev tun10
```

The word "dev" may be omitted.

Bring a link up or down

```
ip link set dev ${interface name} up

ip link set dev ${interface name} down
```

Examples:

```
ip link set dev eth0 down

ip link set dev br0 up
```

Note: virtual links described below, like VLANs and bridges are in **down** state immediately after creation. You need to bring them up to start using them.

Set human-readable link description

```
ip link set dev ${interface name} alias "${description}"
```

Examples:

```
ip link set dev eth0 alias "LAN interface"
```

Link aliases show up in "ip link show" output, like:

```
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DEFAULT qlen
1000
    link/ether 22:ce:e0:99:63:6f brd ff:ff:ff:ff:ff:ff
    alias LAN interface
```

Rename an interface

```
ip link set dev ${old interface name} name ${new interface name}
```

Examples:

```
ip link set dev eth0 name lan
```

Note that you can't rename an active interface. You need to bring it down before doing it.

Change link layer address (usually MAC address)

```
ip link set dev ${interface name} address ${address}
```

Link layer address is a pretty broad concept. The most known example is MAC address for ethernet devices. To change MAC address you would need something like:

```
ip link set dev eth0 address 22:ce:e0:99:63:6f
```

Change link MTU

```
ip link set dev ${interface name} mtu ${MTU value}
```

Examples:

```
ip link set dev tun0 mtu 1480
```

MTU stands for "Maximum Transmission Unit", the maximum size of a frame an interface can transmit at once.

Apart from reducing fragmentation in tunnels like in example above, this is also used to increase performance of gigabit ethernet links that support so called "jumbo frames" (frames up to 9000 bytes large). If all your equipment supports gigabit ethernet, you may want to do something like

```
ip link set dev eth0 mtu 9000
```

Note that you may need to configure it on your L2 switches too, some of them have it disabled by default.

Delete a link

```
ip link delete dev ${interface name}
```

Obviously, only virtual links like VLANs or bridges can be deleted.

Enable or disable multicast on an interface

```
ip link set ${interface name} multicast on

ip link set ${interface name} multicast off
```

Unless you really understand what you are doing, better not to touch this.

Enable or disable ARP on an interface

```
ip link set ${interface name} arp on

ip link set ${interface name} arp off
```

One may want to disable ARP to enforce a security policy and allow only specific MACs to communicate with the interface. In this case neighbor table entries for whitelisted MACs should be created manually (see neighbor table management #Neighbor (ARP and NDP) tables management section), or nothing will be able to communicate with that interface.

In most cases it's better to configure MAC policy on an access layer switch though. Do not change this flag unless you are sure what you are going to do and why.

Create a VLAN interface

```
ip link add name ${VLAN interface name} link ${parent interface name} type vlan id
${tag}
```

Examples:

```
ip link add name eth0.110 link eth0 type vlan id 110
```

The only type of VLAN supported in Linux is IEEE 802.1q VLAN, legacy implementations like ISL are not supported.

Once you create a VLAN interface, all frames tagged with ${tag} you specified in id option received by ${parent interface} will be processed by that VLAN interface.

eth0.100 name format is traditional, but not required, you can name the interface as you want, just like with other interface types.

VLANs can be created over bridge, bonding and other interfaces capable of processing ethernet frames too.

Create a QinQ interface (VLAN stacking)

```
ip link add name ${service interface} link ${physical interface} type vlan proto 802.1ad
id ${service tag}

ip link add name ${client interface} link ${service interface} type vlan proto 802.1q id
${client tag}
```

Example:

```
ip link add name eth0.100 link eth0 type vlan proto 802.1ad id 100 # Create service tag
interface

ip link add name eth0.100.200 link eth0.100 type vlan proto 802.1q id 200 # Create
client tag interface
```

VLAN stacking (aka 802.1ad QinQ) is a way to transmit VLAN tagged traffic over another VLAN. The common use case for it is like this: suppose you are a service provider and you have a customer who wants to use your network infrastructure to connect parts of their network to each other. They use multiple VLANs in their network, so an ordinary rented VLAN is not an option. With QinQ you can add a second tag to the customer traffic when it enters your network and remove that tag when it exits, so there are no conflicts and you don't need to waste VLAN numbers.

The service tag is the VLAN tag the provider uses to carry client traffic through their network. The client tag is the tag set by the customer.

Note that link MTU for the client VLAN interface is not adjusted automatically, you need to take care of it yourself and either decrease the client interface MTU by at least 4 bytes, or increase the parent MTU accordingly.

Standards-compliant QinQ is available since Linux 3.10.

Create pseudo-ethernet (aka macvlan) interface

```
ip link add name ${macvlan interface name} link ${parent interface} type macvlan
```

Examples:

```
ip link add name peth0 link eth0 type macvlan
```

You can think of macvlan interfaces as additional virtual MAC addresses on the parent interface. They look like normal ethernet interfaces from user point of view, and handle all traffic for MAC address they are assigned with received by their parent interface.

This is commonly used for testing, or for using several instances of a service identified by MAC when only one physical interface is available.

They also can be used just for IP address separation instead of assigning multiple addresses to the same physical interface, especially if some service can't operate on a secondary address properly.

Create a dummy interface

```
ip link add name ${dummy interface name} type dummy
```

Examples:

```
ip link add name dummy0 type dummy
```

Dummy interfaces work pretty much like loopback interfaces, just there can be as many of them as you want.

The first purpose of them is for communication of programs inside the host.

The second purpose exploits the fact they are always up (unless administratively taken down). This is often used to assign service addresses to them on routers with more than one physical interface. As long as the traffic to the address assigned to a loopback or dummy interface is routed to the machine that owns it, you can access it through any of its interfaces.

Create a bridge interface

```
ip link add name ${bridge name} type bridge
```

Examples:

```
ip link add name br0 type bridge
```

Bridge interfaces are virtual ethernet switches. They can be used to relay traffic transparently between ethernet interfaces, and, increasingly common, as ethernet switches for virtual machines running inside hypervisors.

You can assign an IP address to a bridge and it will be visible from all bridge ports.

If this command fails, check if "bridge" module is loaded.

Add an interface to bridge

```
ip link set dev ${interface name} master ${bridge name}
```

Examples:

```
ip link set dev eth0 master br0
```

Interface you added to a bridge becomes a virtual switch port. It operates only on datalink layer and ceases all network layer operation.

Remove interface from bridge

```
ip link set dev ${interface name} nomaster
```

Examples:

```
ip link set dev eth0 nomaster
```

Create a bonding interface

```
ip link add name ${name} type bond
```

Examples:

```
ip link add name bond1 type bond
```

Note: This is not enough to configure bonding (link aggregation) in any meaningful way. You need to set up bonding parameters according to your situation. This is far beyond the cheat sheet scope, so consult the documentation.

Interfaces are added to the bond group the same way to bridge group, just note that you can't add it until you take it down.

Create an intermediate functional block interface

```
ip link add ${interface name} type ifb
```

Example:

```
ip link add ifb10 type ifb
```

Intermediate functional block devices are used for traffic redirection and mirroring in conjunction with tc. This is also far beyond the scope of this document, consult tc documentation.

Create a pair of virtual ethernet devices

Virtual ethernet (veth) devices always come in pairs and work as a bidirectional pipe, whatever comes into one of them, comes out of another. They are used in conjunction with system partitioning features such as network namespaces and containers (OpenVZ and LXC) for connecting one partition to another.

```
ip link add name ${first device name} type veth peer name ${second device name}
ip link add name ${first device name} type veth peer name ${second device name}
```

Examples:

```
ip link add name veth-host type veth peer name veth-guest
```

Note: virtual ethernet devices are created in UP state, no need to bring them up manually after creation.

Link group management

Link groups are similar to port ranges found in managed switches. You can add network interfaces to a numbered group and perform operations on all the interfaces from that group at once.

Links not assigned to any group belong to group 0 aka "default".

Add an interface to a group

```
ip link set dev ${interface name} group ${group number}
```

Examples:

```
ip link set dev eth0 group 42
ip link set dev eth1 group 42
```

Remove an interface from a group

This can be done by assigning it to the default group.

```
ip link set dev ${interface name} group 0
ip link set dev ${interface} group default
```

Examples:

```
ip link set dev tun10 group 0
```

Assign a symbolic name to a group

Group names are stored in /etc/iproute2/group file. Symbolic name "default" for group 0 comes exactly from there. You can add your own, one per line, following the same "${number} ${name}" format. You can have up to 255 named groups.

Once you configured a group name, number and name can be used interchangeably in ip commands.

Example:

```
echo "10    customer-vlans" >> /etc/iproute2/group
```

After that you can use that name in all operations, like in

```
ip link set dev eth0.100 group customer-vlans
```

Perform an operation on a group

```
ip link set group ${group number} ${operation and arguments}
```

Examples:

```
ip link set group 42 down
ip link set group uplinks mtu 1200
```

View information about links from specific group

Use usual information viewing command with "group ${group}" modifier.

Examples:

```
ip link list group 42
```

```
ip address show group customers
```

Tun and Tap devices

Tun and tap devices allow userspace programs to emulate a network device. When the userspace program opens them they get a file descriptor. Packets routed by the kernel networking stack to the device are read from the file descriptor, data the userspace program writes to the file descriptor are injected as local outgoing packets into the networking stack. The difference between the two is:

- tap sends and receives raw Ethernet frames.
- tun sends and receives raw IP packets.

The commands listed here manipulate persistent tun/tap devices. There is another sort. Transient tun/tap devices are created by the userspace program when they open a special device, and are destroyed automatically when the associated file descriptor is closed.

Add an tun/tap device useable by root

```
ip tuntap add dev ${interface name} mode ${mode}
```

Examples:

```
ip tuntap add dev tun0 mode tun
```

```
ip tuntap add dev tap9 mode tap
```

Add an tun/tap device usable by an ordinary user

```
ip tuntap add dev ${interface name} mode ${mode} user ${user} group ${group}
```

Example:

```
ip tuntap add dev tun1 mode tun user me group mygroup
```

```
ip tuntap add dev tun2 mode tun user 1000 group 1001
```

Add an tun/tap device using an alternate packet format

Add meta information to each packet received over the file descriptor. Very few programs expect this information, and including it when it isn't expected will break things.

```
ip tuntap add dev ${interface name} mode ${mode} pi
```

Example:

```
ip tuntap add dev tun1 mode tun pi
```

Add an tun/tap ignoring flow control

Normally packets sent to a tun/tap device travel in the same way as packets sent to any other device: they are put on a queue handled by the traffic control engine (which is configured by the tc command). This can be bypassed, thus disabling the traffic control engine for this tun/tap device.

```
ip tuntap add dev ${interface name} mode ${mode} one_queue
```

Example:

```
ip tuntap add dev tun1 mode tun one_queue
```

Delete tun/tap device

```
ip tuntap del dev ${interface name}
```

Examples:

```
ip tuntap del dev tun0
```

Neighbor (ARP and NDP) tables management

For ladies and gentlemen who prefer UK spelling, this command family supports "neighbour" spelling too.

View neighbor tables

```
ip neighbor show
```

All "show" commands support -4 and -6 options to view only IPv4 (ARP) or IPv6 (NDP) neighbors. By default all neighbors are displayed.

View neighbors for single interface

```
ip neighbor show dev ${interface name}
```

Examples:

```
ip neighbor show dev eth0
```

Flush table for an interface

```
ip neighbor flush dev ${interface name}
```

Examples:

```
ip neighbor flush dev eth1
```

Add a neighbor table entry

```
ip neighbor add ${network address} lladdr ${link layer address} dev ${interface name}
```

Examples:

```
ip neighbor add 192.0.2.1 lladdr 22:ce:e0:99:63:6f dev eth0
```

One of the use cases for it is to add static entry for an interface with disabled ARP to restrict interface usage only by hosts with specific MAC addresses.

Delete a neighbor table entry

```
ip neighbor delete ${network address} lladdr ${link layer address} dev ${interface name}
```

Examples:

```
ip neighbor delete 192.0.2.1 lladdr 22:ce:e0:99:63:6f dev eth0
```

Allows to delete a static entry, or get rid of an automatically learnt entry without flushing the table.

Allows to delete a static entry, or get rid of an automatically learnt entry without flushing the table.

Tunnel management

Tunnels are "network wormholes" that look like normal interfaces, but packets sent through them are encapsulated into another protocol and sent to the other side of tunnel through multiple hosts, then decapsulated and processed in usual way, so you can pretend two machines have direct connectivity, while they in fact do not.

This is often used for virtual private networks (in conjunction with encrypted transport protocols like IPsec), or connecting networks that use some protocol via an intermediate network that does not use it (e.g. IPv6 networks separated by an IPv4-only segment).

Note: tunnels on their own offer zero security. They are as secure as their underlying network. So if you need security, use them over an encrypted transport, e.g. IPsec.

Linux currently supports IPIP (IPv4 in IPv4), SIT (IPv6 in IPv4), IP6IP6 (IPv6 in IPv6), IPIP6 (IPv4 in IPv6), GRE (virtually anything in anything), and, in very recent versions, VTI (IPv4 in IPsec).

Note that tunnels are created in DOWN state, you need to bring them up.

In this section ${local endpoint address} and ${remote endpoint address} refer to addresses assigned to physical interfaces of endpoint. ${address} refers to the address assigned to tunnel interface.

Create an IPIP tunnel

```
ip tunnel add ${interface name} mode ipip local ${local endpoint address} remote
${remote endpoint address}
```

Examples:

```
ip tunnel add tun0 mode ipip local 192.0.2.1 remote 198.51.100.3

ip link set dev tun0 up

ip address add 10.0.0.1/30 dev tun0
```

Create a SIT tunnel

```
sudo ip tunnel add ${interface name} mode sit local ${local endpoint address} remote
${remote endpoint address}
```

Examples:

```
ip tunnel add tun9 mode sit local 192.0.2.1 remote 198.51.100.3

ip link set dev tun9 up

ip address add 2001:db8:1::1/64 dev tun9
```

This type of tunnels is commonly used to provide an IPv4-connected network with IPv6 connectivity. There are so called "tunnel brokers" that provide it to everyone interested, e.g. Hurricane Electric tunnelbroker.net http://tunnelbroker.net .

Create an IPIP6 tunnel

```
 ip -6 tunnel add ${interface name} mode ipip6 local ${local endpoint address} remote
${remote endpoint address}
```

Examples:

```
ip -6 tunnel add tun8 mode ipip6 local 2001:db8:1::1 remote 2001:db8:1::2
```

This type of tunnels will be widely used when transit operators phase IPv4 out (i.e. not any soon).

Create an IP6IP6 tunnel

```
ip -6 tunnel add ${interface name} mode ip6ip6 local ${local endpoint address} remote
${remote endpoint address}
```

Examples:

```
ip -6 tunnel add tun3 mode ip6ip6 local 2001:db8:1::1 remote 2001:db8:1::2

ip link set dev tun3 up

ip address add 2001:db8:2:2::1/64 dev tun3
```

Just like IPIP6 these ones aren't going to be generally useful any soon.

Create a gretap (ethernet over GRE) device

```
ip link add ${interface name} type gretap local ${local endpoint address} remote
${remote endpoint address}
```

Examples:

```
ip link add gretap0 type gretap local 192.0.2.1 remote 203.0.113.3
```

This type of tunnels encapsulates ethernet frames into IPv4 packets.

Recent kernel and iproute2 versions also support gretap over IPv6, you need to replace the mode with "ip6gretap" to create an IPv6-based link.

This probably should have been in "Links management" section, but as it involves encapsulation, it's here. Tunnel interface created this way looks like an L2 link, and it can be added to a bridge group. This is used to connect L2 segments via a routed network.

Create a GRE tunnel

```
ip tunnel add ${interface name} mode gre local ${local endpoint address} remote ${remote
endpoint address}
```

Examples:

```
ip tunnel add tun6 mode gre local 192.0.2.1 remote 203.0.113.3

ip link set dev tun6 up

ip address add 192.168.0.1/30 dev tun6

ip address add 2001:db8:1::1/64 dev tun6
```

GRE can encapsulate both IPv4 and IPv6 at the same time. However, by default it uses IPv4 for transport, for GRE over IPv6 there is a separate tunnel mode, "ip6gre".

Create multiple GRE tunnels to the same endpoint

```
ip tunnel add ${interface name} mode gre local ${local endpoint address} remote ${remote
endpoint address} key ${key value}
```

Examples:

```
ip tunnel add tun4 mode gre local 192.0.2.1 remote 203.0.113.6 key 123

ip tunnel add tun5 mode gre local 192.0.2.1 remote 203.0.113.6 key 124
```

Keyed tunnels can be used at the same time to unkeyed too. Key may be in dotted decimal IPv4-like format.

Note that key does not add any security to the tunnel. It's just an identifier used to distinguish one tunnel from another.

Create a point-to-multipoint GRE tunnel

```
ip tunnel add ${interface name} mode gre local ${local endpoint address} key ${key
value}
```

Examples:

```
ip tunnel add tun8 mode gre local 192.0.2.1 key 1234
```

```
ip link set dev tun8 up
```

```
ip address add 10.0.0.1/27 dev tun8
```

Note the absence of ${remote endpoint address}. This is the same to what is called "mode gre multipoint" in Cisco IOS.

In the absence of remote endpoint address the key is the only way to identify the tunnel traffic, so ${key value} is required.

This type of tunnels allows you to communicate with multiple endpoints by using the same tunnel interface. It's commonly used in complex VPN setups with multiple endpoints communicating to one another (in Cisco terminology, "dynamic multipoint VPN").

As there is no explicit remote endpoint address, obviously it is not enough to just create a tunnel. Your system needs to know where the other endpoints are.

In real life NHRP (Next Hop Resolution Protocol) is used for it. For testing you can add peers manually (given remote endpoint uses 203.0.113.6 address on its physical interface and 10.0.0.2 on the tunnel):

```
ip neighbor add 10.0.0.2 lladdr 203.0.113.6 dev tun8
```

You will have to do it on the remote endpoint too, like:

```
ip neighbor add 10.0.0.1 lladdr 192.0.2.1 dev tun8
```

Note that link-layer address and neighbor address are both IP addresses, so they are on the same OSI layer. This one of the cases where link-layer address concept gets interesting.

Create a GRE tunnel over IPv6

Recent kernel and iproute2 versions support GRE over IPv6. Point-to-point with no key:

```
ip -6 tunnel add name ${interface name} mode ip6gre local ${local endpoint} remote
${remote endpoint}
```

It should support all options and features supported by the IPv4 GRE described above.

Delete a tunnel

```
ip tunnel del ${interface name}
```

Examples:

```
ip tunnel del gre1
```

Note that in older iproute2 versions this command did not support the full "delete" word, only "del". Recent versions allow both full and abbreviated forms (tested in iproute2-ss131122).

Modify a tunnel

```
ip tunnel change ${interface name} ${options}
```

Examples:

```
ip tunnel change tun0 remote 203.0.113.89
```

```
ip tunnel change tun10 key 23456
```

Note: Apparently you can't add a key to previously unkeyed tunnel. Not sure if it's a bug or a feature. Also, you can't change tunnel mode on the fly, for obvious reasons.

View tunnel information

```
ip tunnel show
```

```
ip tunnel show ${interface name}
```

Examples:

```
$ip tun show tun99
tun99: gre/ip  remote 10.46.1.20  local 10.91.19.110  ttl inherit
```

L2TPv3 pseudowire management

L2TPv3 http://tools.ietf.org/html/rfc3931 is a tunneling protocol commonly used for L2 pseudowires.

In many distros L2TPv3 is compiled as a module, and may not be loaded by default. If you get a "RTNETLINK answers: No such file or directory" and "Error talking to the kernel" message to any "ip l2tp" command, this is likely the case. Load **l2tp_netlink** and **l2tp_eth** modules. If you want to use L2TPv3 over IP rather than UDP, also load **l2tp_ip**.

Compared to other tunneling protocol implementations in Linux, L2TPv3 terminology is somewhat reversed. You create a *tunnel*, and then bind *sessions* to it. You can bind multiple sessions with different identifiers to the same tunnel. Virtual network interfaces (by default named l2tpethX) are associated with *sessions*.

Note: Linux kernel implements only handling of data frames, so you can create only unmanaged tunnels with iproute2, with all settings configured manually on both sides. If you want to use L2TP for remote access VPN or something else other than fixed pseudowire, you need a userspace daemon to handle it. This is outside of this document scope.

Create an L2TPv3 tunnel over UDP

```
ip l2tp add tunnel \
tunnel_id ${local tunnel numeric identifier} \
peer_tunnel_id ${remote tunnel numeric identifier} \
udp_sport ${source port} \
udp_dport ${destination port} \
encap udp \
local ${local endpoint address} \
remote ${remote endpoint address}
```

Examples:

```
ip l2tp add tunnel \
tunnel_id 1 \
peer_tunnel_id 1 \
udp_sport 5000 \
udp_dport 5000 \
encap udp \
local 192.0.2.1 \
remote 203.0.113.2
```

Note: Tunnel identifiers and other settings on both endpoints must match.

Create an L2TPv3 tunnel over IP

Create an L2TPv3 tunnel over IP

```
ip l2tp add tunnel \
tunnel_id ${local tunnel numeric identifier} \
peer_tunnel_id {remote tunnel numeric identifier } \
encap ip \
local 192.0.2.1 \
remote 203.0.113.2
```

L2TPv3 encapsulated directly into IP offers less overhead, bug generally is unable to pass through NAT.

Create an L2TPv3 session

```
ip l2tp add session tunnel_id ${local tunnel identifier} \
session_id ${local session numeric identifier} \
peer_session_id ${remote session numeric identifier}
```

Examples:

```
ip l2tp add session tunnel_id 1 \
session_id 10 \
peer_session_id 10
```

Notes: tunnel_id value must match a value of previously created tunnel. Session identifiers on both endpoints must match.

Once you create a tunnel and a session, l2tpethX interface will appear, in down state. Change the state to up and bridge it with another interface or assign an address.

Delete an L2TPv3 session

```
ip l2tp del session tunnel_id ${tunnel identifier} \
session_id ${session identifier}
```

Examples

```
ip l2tp del session tunnel_id 1 session_id 1
```

Delete an L2TPv3 tunnel

```
ip l2tp del tunnel tunnel_id ${tunnel identifier}
```

Examples

```
ip l2tp del tunnel tunnel_id 1
```

Note: You need to delete all sessions associated with a tunnel before deleting it.

View L2TPv3 tunnel information

```
ip l2tp show tunnel

ip l2tp show tunnel tunnel_id ${tunnel identifier}
```

Examples:

```
ip l2tp show tunnel tunnel_id 12
```

View L2TPv3 session information

```
ip l2tp show session

ip l2tp show session session_id ${session identifier} \
tunnel_id ${tunnel identifier}
```

Examples:

```
ip l2tp show session session_id 1 tunnel_id 12
```

Policy-based routing

Policy-based routing (PBR) in Linux is designed the following way: first you create custom routing tables, then you create rules to tell the kernel it should use those tables instead of the default table for specific traffic.

Some tables are predefined:

local (table 255)
Contains control routes local and broadcast addresses.

main (table 254)
Contains all non-PBR routes. If you don't specify the table when adding a route, it goes here.

default (table 253)
Reserved for post processing, normally unused.

User-defined tables are created automatically when you add the first route to them.

Create a policy route

```
ip route add ${route options} table ${table id or name}
```

Examples:

```
ip route add 192.0.2.0/27 via 203.0.113.1 table 10

ip route add 0.0.0.0/0 via 192.168.0.1 table ISP2

ip route add 2001:db8::/48 dev eth1 table 100
```

Notes: You can use any route options described in "Route management" section in policy routes too, the only difference is the "table ${table id/name}" part at the end.

Numeric table identifiers and names can be used interchangeably. To create your own symbolic names, edit **/etc/iproute2/rt_tables** config file.

"delete", "change", "replace", or any other route actions work with any table too.

"ip route ... table main" or "ip route ... table 254" would have exact same effect to commands without a table part.

View policy routes

```
ip route show table ${table id or name}
```

Examples:

```
ip route show table 100
```

```
ip route show table test
```

Note: in this case you need the "show" word, the shorthand like "ip route table 120" do not work because the command would be ambiguous.

General rule syntax

```
ip rule add ${options} <lookup ${table id or name}|blackhole|prohibit|unreachable>
```

Traffic that matches the ${options} (described below) will be routed according to the table with specified name/id instead of the "main"/254 table if "lookup" action is used.

"blackhole", "prohibit", and "unreachable" actions that work the same way to route types with same names. In most of examples we will use "lookup" action as the most common.

For IPv6 rules, use "ip -6", the rest of the syntax is the same.

"table ${table id or name}" can be used as alias to "lookup ${table id or name}".

Create a rule to match a source network

```
ip rule add from ${source network} ${action}
```

Examples:

```
ip rule add from 192.0.2.0/24 lookup 10
```

```
ip -6 rule add from 2001:db8::/32 prohibit
```

Notes: "all" can be used as shorthand to 0.0.0.0/0 or ::/0

Create a rule to match a destination network

```
ip rule add to ${destination network} ${action}
```

Examples:

```
ip rule add to 192.0.2.0/24 blackhole
ip -6 rule add to 2001:db8::/32 lookup 100
```

Create a rule to match a ToS field value

```
ip rule add tos ${ToS value} ${action}
```

Examples:

```
ip rule add tos 0x10 lookup 110
```

Create a rule to match a firewall mark value

```
ip rule add fwmark ${mark} ${action}
```

Examples:

```
ip rule add fwmark 0x11 lookup 100
```

Note: See iptables documentation to find out how to set the mark.

Create a rule to match inbound interface

```
ip rule add iif ${interface name} ${action}
```

Examples:

```
ip rule add iif eth0 lookup 10
```

```
ip rule add iif lo lookup 20
```

Rule with "iif lo" (loopback) will match locally generated traffic.

Create a rule to match outbound interface

```
ip rule add oif ${interface name} ${action}
```

Examples:

```
ip rule add oif eth0 lookup 10
```

Note: this works only for locally generated traffic.

Set rule priority

```
ip rule add ${options} ${action} priority ${value}
```

Examples:

```
ip rule add from 192.0.2.0/25 lookup 10 priority 10
```

```
ip rule add from 192.0.2.0/24 lookup 20 priority 20
```

Note: As rules are traversed from the lowest to the highest priority and processing stops at first match, you need to put more specific rules before less specific. The above example demonstrates rules for 192.0.2.0/24 and its subnet 192.0.2.0/25. If the priorities were reversed and the rule for /25 was placed after the rule for /24, it would never be reached.

Show all rules

```
ip rule show
```

```
ip -6 rule show
```

Delete a rule

```
ip rule del ${options} ${action}
```

Examples:

```
ip rule del 192.0.2.0/24 lookup 10
```

Notes: You can copy/paste from the output of "ip rule show"/"ip -6 rule show".

Delete all rules

```
ip rule flush
```

```
ip -6 rule flush
```

Notes: this operation is **highly disruptive**. Even if you have not configured any rules, "from all lookup main" rules are initialized by default. On an unconfigured machine you can see this:

```
$ ip rule show
0:      from all lookup local
32766:  from all lookup main
32767:  from all lookup default

$ ip -6 rule show
0:      from all lookup local
32766:  from all lookup main
```

The "from all lookup local" rule is special and cannot be deleted. The "from all lookup main" is not, there may be valid reasons not to have it, e.g. if you want to route only traffic you created explicit rules for. As may be valid reasons not to have it, e.g. if you want to route only traffic you created explicit rules for. As a side effect, if you do "ip rule flush", this rule will be deleted, which will make the system **stop routing any traffic** until you restore your rules.

netconf (sysctl configuration viewing)

View sysctl configuration for all interfaces

```
ip netconf show
```

View sysctl configuration for specific interface

```
ip netconf show dev ${interface}
```

Examples:

```
ip netconf show dev eth0
```

Network namespace management

Network namespaces are isolated network stack instances within a single machine. They can be used for security domain separation, managing traffic flows between virtual machines and so on.

Every namespace is a complete copy of the networking stack with its own interfaces, addresses, routes etc. You can run processes inside a namespace and bridge namespaces to physical interfaces.

Create a namespace

```
ip netns add ${namespace name}
```

Examples:

```
ip netns add foo
```

List existing namespaces

```
ip netns list
```

Delete a namespace

```
ip netns delete ${namespace name}
```

Examples:

```
ip netns delete foo
```

Run a process inside a namespace

```
ip netns exec ${namespace name} ${command}
```

Examples:

```
ip netns exec foo /bin/sh
```

Note: assigning a process to a non-default namespace requires root privileges.

You can run any processes inside a namespace, in particular you can run "ip" itself, commands like in this "ip netns exec foo ip link list" in this section are not a special syntax but simply executing another copy of "ip" in a namespace. You can run an interactive shell inside a namespace as well.

List all processes assigned to a namespace

```
ip netns pids ${namespace name}
```

The output will be a list of PIDs.

Identify process' primary namespace

```
ip netns identify ${pid}
```

Examples:

```
ip netns identify 9000
```

Assign network interface to a namespace

```
ip link set dev ${interface name} netns ${namespace name}

ip link set dev ${interface name} netns ${pid}
```

Examples:

```
ip link set dev eth0.100 netns foo
```

Note: once you assign an interface to a namespace, it disappears from the default namespace and you will have to perform all operations with it via "ip netns exec ${netspace name}", as in "ip netns exec ${netspace name} ip link set dev dummy0 down".

Moreover, when you move an interface to another namespace, it loses all existing configuration such as IP addresses configured on it and goes to DOWN state. You need to bring it back up and reconfigure.

If you specify a PID instead of a namespace name, the interface gets assigned to the primary namespace of the process with that PID. This way you can reassign an interface back to default namespace with e.g. "ip netns exec ${namespace name} ip link set dev ${intf} netns 1" (since init or another process with PID 1 is pretty much guaranteed to be in default namespace).

Connect one namespace to another

This can be done by creating two veth links and assigning them two different namespaces. Suppose you want to connect namespace "foo" to the default namespace.

Create a pair of veth devices:

```
ip link add name veth1 type veth peer name veth2
```

Move veth2 to namespace foo:

```
ip link set dev veth2 netns foo
```

Bring veth2 and add an address in "foo" namespace:

```
ip netns exec foo ip link set dev veth2 up

ip netns exec foo ip address add 10.1.1.1/24 dev veth2
```

Add an address to veth1, which stays in the default namespace:

```
ip address add 10.1.1.2/24 dev veth1
```

Now you can ping 10.1.1.1 which if in foo namespace, and setup routes to subnets configured in other interfaces of that namespace.

If you want switching instead of routing, you can bridge those veth interfaces with other interfaces in corresponding namespaces. Same technique can be used to connect namespaces to physical networks.

Monitor network namespace subsystem events

```
ip netns monitor
```

Displays events such as creation and deletion of namespaces when they occur.

VXLAN management

VXLAN is a layer 2 tunneling protocol that is commonly used in conjunction with virtualization systems such as KVM to connect virtual machines running on different hypervisor nodes to each other and to outside world.

Unlike GRE or L2TPv3 that are point to point, VXLAN replicates some properties of multiple access switched networks by using IP multicast. Also it supports virtual network separation by transmitting a network identifier along with the frame.

The downside is that you will need to use a multicast routing protocol, typically PIM-SM, to get it to work over routed networks.

The underlying encapsulation protocol for VXLAN is UDP.

Create unicast VXLAN link

```
ip link add name ${interface name} type vxlan \
   id <0-16777215> \
   dev ${source interface}\
   remote ${remote endpoint address} \
   local ${local endpoint address} \
   dstport ${VXLAN destination port}
```

Example:

```
ip link add name vxlan0 type vxlan \
   id 42 dev eth0 remote 203.0.113.6 local 192.0.2.1 dstport 4789
```

Note: id options means VXLAN Network Identifier (VNI).

Create multicast VXLAN link

```
ip link add name ${interface name} type vxlan \
   id <0-16777215> \
   dev ${source interface} \
   group ${multicast address} \
   dstport ${VXLAN destination port}
```

Example:

```
ip link add name vxlan0 type vxlan \
   id 42 dev eth0 group 239.0.0.1 dstport 4789
```

After that you need to bring the link up and either bridge it with another interface or assign an address.

Multicast management

Multicast is mostly handled by applications and routing daemons, so there is not much you can and should do manually here. Multicast-related ip commands are mostly useful for debug.

View multicast groups

```
ip maddress show
```

```
ip maddress show ${interface name}
```

Example:

```
$ip maddress show dev lo
1:      lo
        inet  224.0.0.1
        inet6 ff02::1
        inet6 ff01::1
```

Add a link-layer multicast address

You cannot join an IP multicast group manually, but you can add a multicast MAC address (even though it's rarely needed).

```
ip maddress add ${MAC address} dev ${interface name}
```

Example:

```
ip maddress add 01:00:5e:00:00:ab dev eth0
```

View multicast routes

Multicast routes cannot be added manually, so this command can only show multicast routes installed by a routing daemon. It supports the same modifiers to unicast route viewing commands (iif, table, from etc.).

```
ip mroute show
```

Network event monitoring

You can monitor certain network events with iproute2, such as changes in network configuration, routing tables, and ARP/NDP tables.

Monitor all events

You may either call the command without parameters or explicitly specify "all".

```
ip monitor

ip monitor all
```

Monitor specific events

```
ip monitor ${event type}
```

Event type can be:

link
: Link state: interfaces going up and down, virtual interfaces getting created or destroyed etc.

address
: Link address changes.

route
: Routing table changes.

mroute
: Multicast routing changes.

neigh
: Changes in neighbor (ARP and NDP) tables.

When there are distinct IPv4 and IPv6 subsystems, the usual "-4" and "-6" options allow you to display events only for specified protocol. As in:

```
ip -4 monitor route

ip -6 monitor neigh

ip -4 monitor address
```

Read a log file produced by rtmon

iproute2 includes a program called "rtmon" that serves essentially the same purpose, but writes events to a binary log file instead of displaying them. "ip monitor" command allows you to read files created by the program".

```
ip monitor ${event type} file ${path to the log file}
```

rtmon syntax is similar to that of "ip monitor", except event type is limited to link, address, route, and all; and address family is specified in "-family" option.

```
rtmon [-family <inet|inet6>] [<route|link|address|all>] file ${log file path}
```

Contributors:

Content: Nicolas Dichtel, Russel Stuart, Roan Huang.

Grammar, style, typo fixes: Trick van Staveren, powyginanachochla, Nathan Handler, Bhaskar Sarma Upadhyayula, stappers, depereo, fauxm, eri.

Design: elico, alex-eri.

Last modified: 2017 Feb 10.