Task-centered iproute2 user guide

Overview of iproute2

iproute2 is the Linux networking toolkit that replaced net-tools (ifconfig, vconfig, route, arp etc.). Most of the networking functionality is unified in the ip command. There's also tc for managing traffic policies (QoS), and ss (a netstat replacement), but this document is focused on the ip part.

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

Why learn "new" commands?

First, iproute2 is not new at all. It's been a standard Linux tool since the early 2000's and has been included in every [GNU/]Linux distro by default for a long time.

Old-style network utilities like ifconfig and route are still there only for backward compatibility. They are no longer actively developed. Recent versions of many GNU/Linux distributions no longer install them by default.

A lot of networking features cannot be configured with the old tools and that list is only growing: VRF, network namespaces, policy-based routing, and so on. The ip command provides access to all those features. It also offers a more uniform syntax: for example, ip link set dev eno1 nomaster will work with bridge ports, link aggregation groups, and VRF; while the old approach would make you memorize the syntax of three different commands for those tasks.

It's a tool that takes time to learn, but it's a good time investment for every system and network administrator who works with Linux.

About this document

Historically, documentation has been a weak side of iproute2. The official man pages list available options but don't give almost any usage examples. That need has been addressed by third-party documentation.

This document aims to provide a comprehensive but easy to use guide to the ip command. The ip command controls almost all network subsystems of the Linux kernel. Documenting tc in this style would be a separate big project.

The document is *task-centered*: it tells you how to do different tasks using ip instead of listing available subcommands. It was once called a "cheatsheet" for this reason but has long outgrown the size and scope of a cheat sheet.

Contributing

This document is maintained by Daniil Baturin and distributed under CC-BY-SA 4.0—a strong copyleft, free culture license.

Contributions are always welcome; you can find the source files at github.com/dmbaturin/iproute2-cheatsheet.

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

You can also show your support by buying the maintainer a metaphorical coffee.

This document is provided “as is”, without any warranty. The authors are not liable for any damage related to using it. As usual, think before you type, and think twice before hitting the Enter key.

Typographic conventions

Metasyntactic variables are written in a shell-like syntax, ${something}. Optional command parts are in square brackets. Mandatory arguments are in angle brackets.

General notes

All commands that change any settings require root privileges. Commands that just display information generally do not require special privileges.

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

Abbreviating commands

Any ip command can be abbreviated. For example, ip address add 192.0.2.1/24 dev eth0 can be written ip addr a 192.0.2.1/24 dev eth0 or even ip a a 192.0.2.1/24 dev eth0.

In some cases, you can even omit words. For example, show and list words are always fine to omit: ip address is equivalent to ip address show and ip address list.

Note that the abbreviation system is not always consistent. The dev keyword in ip a a 192.0.2.1/24 dev eth0 cannot be abbreviated, even though every other word can be.

This document intentionally gives all commands in their fullest form for better readability. Using abbreviated commands in scripts and documentation isn't a good idea since the intent may not be immediately obvious to the reader.

Scripting considerations

A common complaint about distributions removing ifconfig is that it forces people to rewrite scripts. However, iproute2 is better for scripting since it supports machine-readable output.

It provides the following output options:

-o (--oneline)
Replaces every line break with a backslash. Supported since the earliest versions.
-br (--brief)
Produces a terse, machine-oriented output. Perfect for dissecting with awk/cut. Supported since at least 4.11.
-j (--json)
Produces JSON output, perfect for non-shell scripts. Supports --pretty and --brief options. Supported since at least 4.13.

Here is a comparison of outputs (--json --pretty and --json --brief are omitted to save space):

```
$ ip --oneline address show lo
1: lo    inet 127.0.0.1/8 scope host lo\       valid_lft forever preferred_lft forever
1: lo    inet6 ::1/128 scope host \       valid_lft forever preferred_lft forever

$ ip --brief address show lo
lo               UNKNOWN        127.0.0.1/8 ::1/128

$ ip --json --brief address show lo
[{"ifname": "lo", "operstate": "UNKNOWN", "addr_info": [{"local": "127.0.0.1", "prefixlen":
8},{"local": "::1", "prefixlen": 128}]}, {}, {}]
```

Address management

---

iproute2 accepts both dotted decimal masks and prefix length values. That is, both 192.0.2.10/24 and 192.0.2.10/255.255.255.0 are acceptable formats.

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 add becomes a “primary address”. The primary address of an interface it’s used as

the source address for outgoing packets by default. All additional addresses you set will become secondary addresses.

Add an address with a 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

The label must start with the interface name followed by a colon due to some backward 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. It will add the address correctly but will ignore the label.

Delete an address from an interface

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

An interface name is required—the kernel will not try to automatically guess which interface you want to remove that address from. Such a guess would not always be unambiguous: Linux does allow the same address to be configured on multiple interfaces, and it has valid use cases (in the Cisco world, this is known as "unnumbered interfaces").

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.

Change the primary address

There is no way to swap primary and secondary IPv4 addresses or explicitly set a new primary IPv4 address. Try to always set the primary address first.

If the sysctl variable net.ipv4.conf.${interface}.promote_secondaries is set to 1, when you delete a primary address, the first secondary address becomes primary. You can enable this behaviour globally with net.ipv4.conf.default.promote_secondaries=1.

Note that when promote_secondaries is set to 0, removing a primary address will also remove **all** secondary addresses from its interface. This setting varies between Linux distributions, so be careful to check it before attempting to change a primary address.

Secondary IPv6 addresses are always promoted to primary if a primary address is deleted.

Neighbor (ARP and NDP) table management

This command supports both American (ip neighbor) and British (ip neighbour) spelling variants.

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 a specific 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 use case for it is a form of data link layer security. You can disable ARP on an interface completely and add MAC addresses of authorized devices by hand.

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

Link management

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

Historically, the ip link command could not create tunnels (IPIP, GRE etc.), VXLAN links, or L2TPv3 pseudowires. Starting from at least iproute2 3.16, it could create anything except L2TPv3 interfaces, and this remains true as of iproute2 5.7. A lot of time, old commands for specific interface types are still more convenient to use, though.

Note that the Linux kernel allows arbitrary, even non-ASCII names for network interfaces. It’s better to stick with alphanumeric because userspace programs (like iptables) may not be so forgiving.

Show information about all links

ip link show

ip link list

These commands are equivalent.

Show information about a specific link

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

Examples:

```
ip link show dev eth0
ip link show dev tun10
```

You can omit the dev word.

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 (tunnels, VLANs, etc.) are always created in the “down” state. 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 the ip link show output, like this:

```
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 renaming it.

Change link-layer address (usually MAC address)

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

A link-layer address is a pretty broad concept. The most known example is the MAC address of an Ethernet device. To change a 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, this is also used to increase the 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 jumbo frames disabled by default.

Delete a link

ip link delete dev ${interface name}

Obviously, only virtual links can be deleted, like VLANs, bridges, or tunnels. For physical interfaces, this command has no effect.

Enable or disable multicast on an interface

ip link set ${interface name} multicast on

ip link set ${interface name} multicast off

Unless you really know what you are doing, better don't touch this option.

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, 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 by Linux is IEEE 802.1q VLAN; legacy implementations like ISL are not supported.

You can use any name for a VLAN interface. eth0.110 is a traditional format, but it's not required.

Any Ethernet-like device can be a parent for a VLAN interface: bridge, bonding, L2 tunnels (GRETAP, L2TPv3…).

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 a service tag interface

ip link add name eth0.100.200 link eth0.100 type vlan proto 802.1q id 200 # Create a 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 a VLAN tag the provider uses to carry client traffic through their network. The client tag is a 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 a virtual MAC (MACVLAN) interface

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

Examples: ip link add name peth0 link eth0 type macvlan

MACVLAN interfaces are like "secondary MAC addresses" on a single interface. They look like normal Ethernet interfaces from the user's point of view, and handle all traffic for their MAC address.

This can be used for services and applications that can't handle secondary IP addresses well.

Create a dummy interface

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

Examples: ip link add name dummy0 type dummy

In Linux, for some strange historical reasons, there's only one loopback interface. Dummy interfaces are like loopbacks, but there can be many of them.

They can be used for communication inside a single host. A loopback or a dummy interface is also a good place to assign a management address on a router with multiple physical 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. You can use them to turn a Linux box into a slow L2 switch, or to enable communication between virtual machines on a hypervisor host. Note that turning a Linux box into a physical switch isn't a completely absurd idea, since unlike dumb hardware switches, it can work as a transparent firewall.

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

If bridge creation fails, check if the bridge module is loaded.

Add a bridge port

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

Examples: ip link set dev eth0 master br0

An interface you add to a bridge becomes a virtual switch port. It operates only on the data link layer and ceases all network layer operation.

Delete a bridge port

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.

Bonding interface members are added and removed exactly like bridge ports, with master and nomaster commands.

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 the 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 the other. They are used in conjunction with system partitioning features such as network namespaces and containers (OpenVZ or LXC) for connecting one partition to another.

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 the 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 (“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 configured in the /etc/iproute2/group file. The symbolic name “default” for group 0 isn’t an iproute2 built-in; it comes from the default group config file. 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, its 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 this:

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 a specific group

Use a usual information viewing command with a 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. The difference between the two is that TAP devices work with Ethernet frames (L2 device), while TUN works with IP packets (L3 device).

There are two types of TUN/TAP devices: persistent and transient. Transient TUN/TAP devices are created by userspace programs when they open a special device and are destroyed automatically when the associated file descriptor is closed. Persistent devices are created with ip commands documented below.

View TUN/TAP devices

ip tuntap show

Note: this command can be abbreviated ip tuntap.

This command is the only way to find out if some device is in the TUN or TAP mode.

Add a TUN/TAP device useable by the root user

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

Examples:

ip tuntap add dev tun0 mode tun

ip tuntap add dev tap9 mode tap

Add a 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 a 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 programs that don't expect it will not function correctly with a device in this mode.

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

Example: ip tuntap add dev tun1 mode tun pi

Add a 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 in 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 a TUN/TAP device

ip tuntap delete dev ${interface name} mode ${mode}

Examples:

ip tuntap delete dev tun0 mode tun

ip tuntap delete dev tap1 mode tap

Note: you must specify the mode. The mode is not displayed in ip link show, so if you don't know if it's a TUN or a TAP device, consult the output of ip tuntap show.

Tunnel management

Tunnels are "network wormholes" that emulate a direct connection over a routed network by encapsulating entire packets into another protocol.

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 VTI (IPv4 in IPsec).

Note that tunnels are created in the 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, while ${address} refers to an address assigned to a 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 an SIT (6in4) tunnel

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 tunnel 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’s 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 tunnel will be widely used only 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 widely used any soon.

Create an L2 GRE tunnel device

Static GRE tunnels are traditionally used for encapsulating IPv4 or IPv6 packets, but the RFC does not limit GRE payloads to L3 protocol packets. It's possible to encapsulate anything, including Ethernet frames.

However, in Linux, the gre encapsulation refers specifically to L3 devices, while for an L2 device capable of transmitting Ethernet frames, you need to use the gretap encapsulation.

```
# L2 GRE over IPv4
ip link add ${interface name} type gretap local ${local endpoint address} remote ${remote endpoint address}

# L2 GRE over IPv6
ip link add ${interface name} type ip6gretap local ${local endpoint address} remote ${remote endpoint address}
```

These tunnels can be bridged with other physical and virtual interfaces.

Examples:

ip link add gretap0 type gretap local 192.0.2.1 remote 203.0.113.3

ip link add gretap1 type ip6gretap local 2001:db8:dead::1 remote 2001:db8:beef::2

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

RFC2890 defines “keyed” GRE tunnels. A “key” in this case has nothing to do with encryption; it’s simply an identifier that allows routers to tell one tunnel from another, so you can create multiple tunnels between the same endpoints.

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

You can also specify keys in a dotted-decimal IPv4-like format.

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 as “mode gre multipoint” in Cisco IOS.

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

This type of tunnel 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”).

Since there is no explicit remote endpoint address, it is obviously 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 as well:

ip neighbor add 10.0.0.1 lladdr 192.0.2.1 dev tun8

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 syntax, 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 a 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 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 running any ip l2tp command produces errors like RTNETLINK answers: No such file or directory and Error talking to the kernel, you need to load l2tp_netlink and l2tp_eth kernel modules. If you want to use L2TPv3 over IP rather than over UDP, also load l2tp_ip.

Compared to other tunneling protocol implementations in Linux, L2TPv3 terminology is somewhat backwards. 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* rather than *tunnels*.

Note

You can only create *static* (unmanaged) L2TPv3 tunnels with iproute2. If you want to use L2TP for remote access VPN or otherwise need dynamically created pseudowires, you need a userspace daemon to handle it. That is outside of this document's 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

```
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
```

Encapsulating L2TPv3 frames in IP rather than in UDP creates less overhead, but may cause problems for endpoints behind a 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 the value of an existing tunnel (iproute2 will not create a tunnel if it doesn't exist). Session identifiers on both endpoints must match.

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

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 the tunnel itself.

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`

VXLAN management

VXLAN is a tunneling protocol designed for distributed switched networks. It's often used in virtualization setups to decouple the virtual network topology from that of the underlying physical network.

VXLAN can work in either multicast or unicast mode and supports isolating virtual networks using a VNI (virtual network identifier), similar to VLANs in Ethernet networks.

The downside of the multicast mode is that you will need to use a multicast routing protocol, typically PIM-SM, to get it to work over routed networks, but if you get it set up, you don't need to create all VXLAN connections by hand.

The underlying encapsulation protocol for VXLAN is UDP.

Create a 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: the id option is the VXLAN Network Identifier (VNI).

Create a 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 to it.

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

The Linux kernel does not keep routes with unreachable next hops. If a link goes down, all routes that would use that link are permanently removed from the routing table. You may not have noticed this behaviour because, in many cases, additional software (e.g. NetworkManager or rp-pppoe) takes care of restoring the routes when links go up and down.

If you are going to use your Linux machine as a router, consider installing a routing protocol suite such as FreeRangeRouting or BIRD. They keep track of link states 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 that network (this is why the subnet mask is required). Such 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. The 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. By default, only 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” is optional in all “show” commands.

View routes to a network and all supernets

```
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.

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

View routes to an 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”, since it always shows only 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 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 a 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 an 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 a next hop address is not required.

Change or replace a route

For modifying routes, there are ip route change and ip route replace commands. The difference between them is that the change command will produce an error if you try to change a route that doesn't exist. The replace command will create a route if it doesn't exist already.

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 ${route specifier}

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 for creating default routes.

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}

For IPv6 routes, default 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.

There are two use cases for blackhole routes. First, they can work as a very fast outbound traffic filter, e.g., to make known botnet controllers inaccessible or to protect a server inside your network from an incoming DDoS attack. Second, they can be used to trick a routing protocol daemon into thinking you have a route to a network if you only have real routes to its parts but want to advertise it aggregated.

Other special routes

There are a few other types of special purpose 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 an even greater traffic flow. They can be good for implementing internal access policies, but a firewall is usually a better idea.

“Throw” routes may be used for implementing policy-based routing. In non-default, tables they stop the lookup process 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 kernel prefers the one with the *lowest* metric.

An 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 section), and the system will fall back to routes with higher metric values.

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 2 will get roughly two times more traffic than another one with weight 1). You can have as many gateways as you want, and you can mix gateway and interface routes:

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

Warning: the downside of this type of load 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, this is usually fine, and in some cases even unavoidable.

If your system does some local processing (e.g. NAT), this may cause problems with incoming connections. In that case, you should be using a stateful L4 load balancing setup instead.

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 which tables to use for which packets.

Some tables are predefined:

- local (table 255): Contains control routes to 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 to this table.
- 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

Note: You can use any route options described in the route management section for 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 the /etc/iproute2/rt_tables config file.

delete, change, replace, and all other route actions work with any table too.

ip route ... table main or ip route ... table 254 have the exact same effect as 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; a shorthand like ip route table 120 does 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 the lookup action is used.

blackhole, prohibit, and unreachable actions work just like in the default table.

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

table ${table id or name} can be used as a shortcut for 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 keyword can be used as a shortcut for 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.

Make sure to set the mark in a firewall chain that is processed before the routing decision, else your PBR rules that use that mark will have no effect. You can find an excellent netfilter flowchart in Phil Hagen's blog. For forwarded traffic, mangle FORWARD should be a good place, e.g. iptables -t mangle -I FORWARD -s 192.0.2.1 -j MARK --set-mark 0x11.

Create a rule to match an inbound interface

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

Examples:

ip rule add iif eth0 lookup 10

ip rule add iif lo lookup 20

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

Create a rule to match an 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.

Create a rule to match a user id range

ip rule add uidrange <${start}-${end}>

Examples:

ip rule add uidrange 1000-1100 lookup 10
ip rule add uidrange 3000-3000 lookup 20

To apply a rule to a single user, use the same UID for both the start and end of the range.

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: Rules are traversed from the lowest to the highest priority, and processing stops on the first match, so you need to put more specific rules before less specific ones. The example above 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 or 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, some fundamental rules like from all lookup main rules are created for you 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 a side effect, if you do ip rule flush, that rule will be deleted, which will make the system stop routing any traffic until you restore your rules.

VRF management

VRF (Virtual Routing and Forwarding) is a mechanism for isolating routes of a network in a separate routing table. It allows multiple networks with conflicting address ranges to co-exist in the same router. The most common use case for it is multi-tenant setups and provider-supported VPNs where customers can use their own network addresses.

The main difference from policy-based routing is that “normal” non-default tables used in PBR only separate static routes but not connected routes, so they cannot resolve address conflicts. When a network interface is bound to a VRF, all connected routes from it will be moved to a separate routing table.

Unlike network namespaces, VRFs work exclusively on the network layer. They do not create a separate copy of the network stack, do not interfere with L2 protocols such as LLDP, and one process can bind sockets to multiple VRFs (useful for dynamic routing protocol or IPsec daemons).

Create a VRF

ip link add ${name} type vrf table ${table}

Example: sudo ip link add foo type vrf table 100

If routing table 100 does not exist, it will be automatically created.

View configured VRFs

ip vrf show

```
ip link show vrf ${vrf}
```

The output of `ip vrf show` only shows associations between VRFs and routing tables, but not associations between VRFs and network interfaces. To view all interfaces of a VRF, use `ip link show vrf ${vrf}`.

Bind an interface to a VRF

```
ip link set ${interface} master ${vrf}
```

Example: `ip link set eth0 master foo`.

All connected routes associated with the interface will be moved to the VRF table. For example, if eth0 has address 192.0.2.1/24, and a VRF instance foo uses table 100, then the route to 192.0.2.0/24 will disappear from the main table and re-appear in table 100.

Remove an interface from a VRF

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

Connected routes associated with that interface will be moved back to the main table.

Run a command inside a VRF

```
ip vrf exec ${vrf} ${command}
```

Example: `ip vrf exec ping 192.0.2.100`.

The traffic of the process will be routed according to the VRF table routes, so it's useful for troubleshooting.

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}

Example: ip netns add foo

List existing namespaces

ip netns list

Delete a namespace

ip netns delete ${namespace name}

Example: ip netns delete foo.

Run a process inside a namespace

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

Example: 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 /sbin/ip itself. Commands like ip netns exec foo ip link list in this section are not special: we are 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}

Example: ip netns identify 9000.

Assign a network interface to a namespace

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

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

Example: 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}, like 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 the DOWN state. You'll need to bring it back up and reconfigure it.

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 the 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 a pair of veth links and assigning them to different namespaces.

Suppose you want to connect a namespace named "foo" to the default namespace. First, 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 is in the foo namespace, and set up 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. The same technique can be used for connecting namespaces to physical networks.

Monitor network namespace subsystem events

```
ip netns monitor
```

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

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 debugging.

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 as 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 the specified protocol.

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. You can read those logs file with an ip monitor command.

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

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

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

netconf (sysctl configuration viewing)

View sysctl configuration for all interfaces

ip netconf show

View sysctl configuration for specific interface

ip netconf show dev ${interface}

Example: ip netconf show dev eth0

Contributors

Content: Nicolas Dichtel, Russel Stuart, Phil Huang, Haishan, Emil Pederson, Nick B.

Grammar, style, typo fixes: Trick van Staveren, powyginanachochla, Nathan Handler, Bhaskar Sarma Upadhyayula, Geert Stappers, Alex White-Robinson, Achilleas Pipinellis, fauxm, fgtham, eri, Zhuoyun Wei, Jonathan ZHAO, Julien Barbot, thoastbrot.