IP Command Reference.

Alexey N. Kuznetsov
Institute for Nuclear Research, Moscow
kuznet@ms2.inr.ac.ru
April 14, 1999

Contents

1 About this document

This document presents a comprehensive description of the `ip` utility from the `iproute2` package. It is not a tutorial or user's guide. It is a *dictionary*, not explaining terms, but translating them into other terms, which may also be unknown to the reader. However, the document is self-contained and the reader, provided they have a basic networking background, will find enough information and examples to understand and configure Linux-2.2 IP and IPv6 networking.

This document is split into sections explaining `ip` commands and options, decrypting `ip` output and containing a few examples. More voluminous examples and some topics, which require more elaborate discussion, are in the appendix.

The paragraphs beginning with NB contain side notes, warnings about bugs and design drawbacks. They may be skipped at the first reading.

2 `ip` — command syntax

The generic form of an `ip` command is:

```
ip [ OPTIONS ] OBJECT [ COMMAND [ ARGUMENTS ]]
```

where `OPTIONS` is a set of optional modifiers affecting the general behaviour of the `ip` utility or changing its output. All options begin with the character `'-'` and may be used in either long or abbreviated forms. Currently, the following options are available:

- `-V`, `-Version`

  — print the version of the `ip` utility and exit.

- `-s`, `-stats`, `-statistics`

  — output more information. If the option appears twice or more, the amount of information increases. As a rule, the information is statistics or some time values.

- `-f`, `-family` followed by a protocol family identifier: `inet`, `inet6` or `link`.

  — enforce the protocol family to use. If the option is not present, the protocol family is guessed from other arguments. If the rest of the command line does not give enough information to guess the family, `ip` falls back to the default one, usually `inet` or `any`. `link` is a special family identifier meaning that no networking protocol is involved.

- `-4`

  — shortcut for `-family inet`.

- `-6`

  — shortcut for `-family inet6`.

- `-0`

  — shortcut for `-family link`.

- `-o`, `-oneline`

  — output each record on a single line, replacing line feeds with the `'\'` character. This is convenient when you want to count records with `wc` or to `grep` the output. The trivial script `rtpr` converts the output back into readable form.

- `-r`, `-resolve`

  — use the system's name resolver to print DNS names instead of host addresses.

  > NB. Do not use this option when reporting bugs or asking for advice.

  > NB. `ip` never uses DNS to resolve names to addresses.

`OBJECT` is the object to manage or to get information about. The object types currently understood by `ip` are:

- `link` — network device
- `address` — protocol (IP or IPv6) address on a device
- `neighbour` — ARP or NDISC cache entry
- `route` — routing table entry
- `rule` — rule in routing policy database
- `maddress` — multicast address
- `mroute` — multicast routing cache entry
- `tunnel` — tunnel over IP

Again, the names of all objects may be written in full or abbreviated form, f.e. `address` is abbreviated as `addr` or just `a`.

`COMMAND` specifies the action to perform on the object. The set of possible actions depends on the object type. As a rule, it is possible to `add`, `delete` and `show` (or `list`) objects, but some objects do not allow all of these operations or have some additional commands. The `help` command is available for all objects. It prints out a list of available commands and argument syntax conventions.

If no command is given, some default command is assumed. Usually it is `list` or, if the objects of this class cannot be listed, `help`.

`ARGUMENTS` is a list of arguments to the command. The arguments depend on the command and object. There are two types of arguments: *flags*, consisting of a single keyword, and *parameters*, consisting of a keyword followed by a value. For convenience, each command has some *default parameter* which may be omitted. F.e. parameter `dev` is the default for the `ip link` command, so `ip link ls eth0` is equivalent to `ip link ls dev eth0`. In the command descriptions below such parameters are distinguished with the marker: "(default)".

Almost all keywords may be abbreviated with several first (or even single) letters. The shortcuts are convenient when `ip` is used interactively, but they are not recommended in scripts or when reporting bugs or asking for advice. "Officially" allowed abbreviations are listed in the document body.

3 ip — error messages

`ip` may fail for one of the following reasons:

- A syntax error on the command line: an unknown keyword, incorrectly formatted IP address *et al*. In this case `ip` prints an error message and exits. As a rule, the error message will contain information about the reason for the failure. Sometimes it also prints a help page.
- The arguments did not pass verification for self-consistency.
- `ip` failed to compile a kernel request from the arguments because the user didn't give enough information.
- The kernel returned an error to some syscall. In this case `ip` prints the error message, as it is output with `perror(3)`, prefixed with a comment and a syscall identifier.
- The kernel returned an error to some RTNETLINK request. In this case `ip` prints the error message, as it is output with `perror(3)` prefixed with "RTNETLINK answers:".

All the operations are atomic, i.e. if the `ip` utility fails, it does not change anything in the system. One harmful exception is `ip link` command (Sec.4, p.6), which may change only some of the device parameters given on command line.

It is difficult to list all the error messages (especially syntax errors). However, as a rule, their meaning is clear from the context of the command.

The most common mistakes are:

1. Netlink is not configured in the kernel. The message is:

```
Cannot open netlink socket: Invalid value
```

2. RTNETLINK is not configured in the kernel. In this case one of the following messages may be printed, depending on the command:

```
Cannot talk to rtnetlink: Connection refused
Cannot send dump request: Connection refused
```

3. The `CONFIG_IP_MULTIPLE_TABLES` option was not selected when configuring the kernel. In this case any attempt to use the `ip rule` command will fail, f.e.

```
kuznet@kaiser $ ip rule list
RTNETLINK error: Invalid argument
dump terminated
```

4 `ip link` — network device configuration

Object: A `link` is a network device and the corresponding commands display and change the state of devices.

Commands: `set` and `show` (or `list`).

4.1 `ip link set` — change device attributes

Abbreviations: `set`, `s`.

Arguments:

- `dev NAME` (default)

  — `NAME` specifies the network device on which to operate.

- `up` and `down`

  — change the state of the device to `UP` or `DOWN`.

- `arp on` or `arp off`

  — change the `NOARP` flag on the device.

  > NB. This operation is *not allowed* if the device is in state `UP`. Though neither the `ip` utility nor the kernel check for this condition. You can get unpredictable results changing this flag while the device is running.

- `multicast on` or `multicast off`

  — change the `MULTICAST` flag on the device.

- `dynamic on` or `dynamic off`

  — change the `DYNAMIC` flag on the device.

- `name NAME`

  — change the name of the device. This operation is not recommended if the device is running or has some addresses already configured.

- `txqueuelen NUMBER` or `txqlen NUMBER`

  — change the transmit queue length of the device.

- `mtu NUMBER`

  — change the MTU of the device.

- `address LLADDRESS`

  — change the station address of the interface.

- `broadcast LLADDRESS`, `brd LLADDRESS` or `peer LLADDRESS`

  — change the link layer broadcast address or the peer address when the interface is `POINTOPOINT`.

  > NB. For most devices (f.e. for Ethernet) changing the link layer broadcast address will break networking. Do not use it, if you do not understand what this operation really does.

NB. The `ip` utility does not change the `PROMISC` or `ALLMULTI` flags. These flags are considered obsolete and should not be changed administratively.

Warning: If multiple parameter changes are requested, `ip` aborts immediately after any of the changes have failed. This is the only case when `ip` can move the system to an unpredictable state. The solution is to avoid changing several parameters with one `ip link set` call.

Examples:

- `ip link set dummy address 00:00:00:00:00:01`

  — change the station address of the interface `dummy`.

- `ip link set dummy up`

  — start the interface `dummy`.

4.2 `ip link show` — display device attributes

Abbreviations: `show`, `list`, `lst`, `sh`, `ls`, `l`.

Arguments:

- `dev NAME` (default)

  — `NAME` specifies the network device to show. If this argument is omitted all devices are listed.

- `up`

  — only display running interfaces.

Output format:

```
kuznet@alisa:~ $ ip link ls eth0
3: eth0: <BROADCAST,MULTICAST,UP> mtu 1500 qdisc cbq qlen 100
    link/ether 00:a0:cc:66:18:78 brd ff:ff:ff:ff:ff:ff
kuznet@alisa:~ $ ip link ls sit0
5: sit0@NONE: <NOARP,UP> mtu 1480 qdisc noqueue
    link/sit 0.0.0.0 brd 0.0.0.0
kuznet@alisa:~ $ ip link ls dummy
2: dummy: <BROADCAST,NOARP> mtu 1500 qdisc noop
    link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
kuznet@alisa:~ $
```

The number before each colon is an *interface index* or *ifindex*. This number uniquely identifies the interface. This is followed by the *interface name* (`eth0`, `sit0` etc.). The interface name is also unique at every given moment. However, the interface may disappear from the list (f.e. when the corresponding driver module is unloaded) and another one with the same name may be created later. Besides that, the administrator may change the name of any device with `ip link set name` to make it more intelligible.

The interface name may have another name or `NONE` appended after the `@` sign. This means that this device is bound to some other device, i.e. packets send through it are encapsulated and sent via the "master" device. If the name is `NONE`, the master is unknown.

Then we see the interface *mtu* ("maximal transfer unit"). This determines the maximal size of data which can be sent as a single packet over this interface.

qdisc ("queuing discipline") shows the queuing algorithm used on the interface. Particularly, `noqueue` means that this interface does not queue anything and `noop` means that the interface is in blackhole mode i.e. all packets sent to it are immediately discarded. *qlen* is the default transmit queue length of the device measured in packets.

The interface flags are summarized in the angle brackets.

- `UP` — the device is turned on. It is ready to accept packets for transmission and it may inject into the kernel packets received from other nodes on the network.

- `LOOPBACK` — the interface does not communicate with other hosts. All packets sent through it will be returned and nothing but bounced packets can be received.

- `BROADCAST` — the device has the facility to send packets to all hosts sharing the same link. A typical example is an Ethernet link.

- `POINTOPOINT` — the link has only two ends with one node attached to each end. All packets sent to this link will reach the peer and all packets received by us came from this single peer.

  If neither `LOOPBACK` nor `BROADCAST` nor `POINTOPOINT` are set, the interface is assumed to be NMBA (Non-Broadcast Multi-Access). This is the most generic type of device and the most complicated one, because the host attached to a NBMA link has no means to send to anyone without additionally configured information.

- `MULTICAST` — is an advisory flag indicating that the interface is aware of multicasting i.e. sending packets to some subset of neighbouring nodes. Broadcasting is a particular case of multicasting, where the multicast group consists of all nodes on the link. It is important to emphasize that software *must not* interpret the absence of this flag as the inability to use multicasting on this interface. Any `POINTOPOINT` and `BROADCAST` link is multicasting by definition, because we have direct access to all the neighbours and, hence, to any part of them. Certainly, the use of high bandwidth multicast transfers is not recommended on broadcast-only links because of high expense, but it is not strictly prohibited.

- `PROMISC` — the device listens to and feeds to the kernel all traffic on the link even if it is not destined for us, not broadcasted and not destined for a multicast group of which we are member. Usually this mode exists only on broadcast links and is used by bridges and for network monitoring.

- `ALLMULTI` — the device receives all multicast packets wandering on the link. This mode is used by multicast routers.

- `NOARP` — this flag is different from the other ones. It has no invariant value and its interpretation depends on the network protocols involved. As a rule, it indicates that the device needs no address resolution and that the software or hardware knows how to deliver packets without any help from the protocol stacks.

- `DYNAMIC` — is an advisory flag indicating that the interface is dynamically created and destroyed.

- `SLAVE` — this interface is bonded to some other interfaces to share link capacities.

NB. There are other flags but they are either obsolete (`NOTRAILERS`) or not implemented (`DEBUG`) or specific to some devices (`MASTER`, `AUTOMEDIA` and `PORTSEL`). We do not discuss them here.

NB. The values of `PROMISC` and `ALLMULTI` flags shown by the `ifconfig` utility and by the `ip` utility are *different*. `ip link ls` shows the true device state, while `ifconfig` shows the virtual state which was set with `ifconfig` itself.

The second line contains information on the link layer addresses associated with the device. The first word (`ether`, `sit`) defines the interface hardware type. This type determines the format and semantics of the addresses and is logically part of the address. The default format of the station address and the broadcast address (or the peer address for pointopoint links) is a sequence of hexadecimal bytes separated by colons, but some link types may have their natural address format, f.e. addresses of tunnels over IP are printed as dotted-quad IP addresses.

NB. NBMA links have no well-defined broadcast or peer address, however this field may contain useful information, f.e. about the address of broadcast relay or about the address of the ARP server.

NB. Multicast addresses are not shown by this command, see `ip maddr ls` in Sec.9 (p.40 of this document).

Statistics: With the `-statistics` option, `ip` also prints interface statistics:

```
kuznet@alisa:~ $ ip -s link ls eth0
3: eth0: <BROADCAST,MULTICAST,UP> mtu 1500 qdisc cbq qlen 100
    link/ether 00:a0:cc:66:18:78 brd ff:ff:ff:ff:ff:ff
    RX: bytes  packets  errors  dropped overrun mcast
    2449949362 2786187  0       0       0       0
    TX: bytes  packets  errors  dropped carrier collsns
    178558497  1783945  332     0       332     35172
kuznet@alisa:~ $
```

`RX:` and `TX:` lines summarize receiver and transmitter statistics. They contain:

- `bytes` — the total number of bytes received or transmitted on the interface. This number wraps when the maximal length of the data type natural for the architecture is exceeded, so continuous monitoring requires a user level daemon snapping it periodically.
- `packets` — the total number of packets received or transmitted on the interface.
- `errors` — the total number of receiver or transmitter errors.
- `dropped` — the total number of packets dropped due to lack of resources.

- `overrun` — the total number of receiver overruns resulting in dropped packets. As a rule, if the interface is overrun, it means serious problems in the kernel or that your machine is too slow for this interface.
- `mcast` — the total number of received multicast packets. This option is only supported by a few devices.
- `carrier` — total number of link media failures f.e. because of lost carrier.
- `collsns` — the total number of collision events on Ethernet-like media. This number may have a different sense on other link types.
- `compressed` — the total number of compressed packets. This is available only for links using VJ header compression.

If the `-s` option is entered twice or more, `ip` prints more detailed statistics on receiver and transmitter errors.

```
kuznet@alisa:~ $ ip -s -s link ls eth0
3: eth0: <BROADCAST,MULTICAST,UP> mtu 1500 qdisc cbq qlen 100
    link/ether 00:a0:cc:66:18:78 brd ff:ff:ff:ff:ff:ff
    RX: bytes  packets  errors  dropped overrun mcast
    2449949362 2786187  0       0       0       0
    RX errors: length   crc     frame   fifo    missed
               0        0       0       0       0
    TX: bytes  packets  errors  dropped carrier collsns
    178558497  1783945  332     0       332     35172
    TX errors: aborted  fifo    window  heartbeat
               0        0       0       332
kuznet@alisa:~ $
```

These error names are pure Ethernetisms. Other devices may have non zero values in these fields but they may be interpreted differently.

5 `ip address` — protocol address management

Abbreviations: `address`, `addr`, `a`.

Object: The `address` is a protocol (IP or IPv6) address attached to a network device. Each device must have at least one address to use the corresponding protocol. It is possible to have several different addresses attached to one device. These addresses are not discriminated, so that the term *alias* is not quite appropriate for them and we do not use it in this document.

The `ip addr` command displays addresses and their properties, adds new addresses and deletes old ones.

Commands: `add`, `delete`, `flush` and `show` (or `list`).

5.1 `ip address add` — add a new protocol address

Abbreviations: `add`, `a`.

Arguments:

- `dev NAME`

  — the name of the device to add the address to.

- `local ADDRESS` (default)

  — the address of the interface. The format of the address depends on the protocol. It is a dotted quad for IP and a sequence of hexadecimal halfwords separated by colons for IPv6. The `ADDRESS` may be followed by a slash and a decimal number which encodes the network prefix length.

- `peer ADDRESS`

  — the address of the remote endpoint for pointopoint interfaces. Again, the `ADDRESS` may be followed by a slash and a decimal number, encoding the network prefix length. If a peer address is specified, the local address *cannot* have a prefix length. The network prefix is associated with the peer rather than with the local address.

- `broadcast ADDRESS`

  — the broadcast address on the interface.

  It is possible to use the special symbols `'+'` and `'-'` instead of the broadcast address. In this case, the broadcast address is derived by setting/resetting the host bits of the interface prefix.

  > NB. Unlike `ifconfig`, the `ip` utility *does not* set any broadcast address unless explicitly requested.

- `label NAME`

  — Each address may be tagged with a label string. In order to preserve compatibility with Linux-2.0 net aliases, this string must coincide with the name of the device or must be prefixed with the device name followed by colon.

- `scope SCOPE_VALUE`

  — the scope of the area where this address is valid. The available scopes are listed in file `/etc/iproute2/rt_scopes`. Predefined scope values are:

  - `global` — the address is globally valid.

  - `site` — (IPv6 only) the address is site local, i.e. it is valid inside this site.
  - `link` — the address is link local, i.e. it is valid only on this device.
  - `host` — the address is valid only inside this host.

Appendix A (p.48 of this document) contains more details on address scopes.

Examples:

- `ip addr add 127.0.0.1/8 dev lo brd + scope host`

  — add the usual loopback address to the loopback device.

- `ip addr add 10.0.0.1/24 brd + dev eth0 label eth0:Alias`

  — add the address 10.0.0.1 with prefix length 24 (i.e. netmask `255.255.255.0`), standard broadcast and label `eth0:Alias` to the interface `eth0`.

5.2 `ip address delete` — delete a protocol address

Abbreviations: `delete`, `del`, `d`.

Arguments: coincide with the arguments of `ip addr add`. The device name is a required argument. The rest are optional. If no arguments are given, the first address is deleted.

Examples:

- `ip addr del 127.0.0.1/8 dev lo`

  — deletes the loopback address from the loopback device. It would be best not to repeat this experiment.

- Disable IP on the interface `eth0`:

```
while ip -f inet addr del dev eth0; do
  : nothing
done
```

  Another method to disable IP on an interface using `ip addr flush` may be found in sec.5.4, p.15.

5.3 `ip address show` — display protocol addresses

Abbreviations: `show`, `list`, `lst`, `sh`, `ls`, `l`.

Arguments:

- dev NAME (default)

  — the name of the device.

- scope SCOPE_VAL

  — only list addresses with this scope.

- to PREFIX

  — only list addresses matching this prefix.

- label PATTERN

  — only list addresses with labels matching the PATTERN. PATTERN is a usual shell style pattern.

- dynamic and permanent

  — (IPv6 only) only list addresses installed due to stateless address configuration or only list permanent (not dynamic) addresses.

- tentative

  — (IPv6 only) only list addresses which did not pass duplicate address detection.

- deprecated

  — (IPv6 only) only list deprecated addresses.

- primary and secondary

  — only list primary (or secondary) addresses.

Output format:

```
kuznet@alisa:~ $ ip addr ls eth0
3: eth0: <BROADCAST,MULTICAST,UP> mtu 1500 qdisc cbq qlen 100
    link/ether 00:a0:cc:66:18:78 brd ff:ff:ff:ff:ff:ff
    inet 193.233.7.90/24 brd 193.233.7.255 scope global eth0
    inet6 3ffe:2400:0:1:2a0:ccff:fe66:1878/64 scope global dynamic
       valid_lft forever preferred_lft 604746sec
    inet6 fe80::2a0:ccff:fe66:1878/10 scope link
kuznet@alisa:~ $
```

The first two lines coincide with the output of ip link ls. It is natural to interpret link layer addresses as addresses of the protocol family AF_PACKET.

Then the list of IP and IPv6 addresses follows, accompanied by additional address attributes: scope value (see Sec.5.1, p.12 above), flags and the address label.

Address flags are set by the kernel and cannot be changed administratively. Currently, the following flags are defined:

1. `secondary`

   — the address is not used when selecting the default source address of outgoing packets (Cf. Appendix A, p.48.). An IP address becomes secondary if another address with the same prefix bits already exists. The first address is primary. It is the leader of the group of all secondary addresses. When the leader is deleted, all secondaries are purged too.

2. `dynamic`

   — the address was created due to stateless autoconfiguration [2]. In this case the output also contains information on times, when the address is still valid. After `preferred_lft` expires the address is moved to the deprecated state. After `valid_lft` expires the address is finally invalidated.

3. `deprecated`

   — the address is deprecated, i.e. it is still valid, but cannot be used by newly created connections.

4. `tentative`

   — the address is not used because duplicate address detection [2] is still not complete or failed.

5.4 `ip address flush` — flush protocol addresses

Abbreviations: `flush`, `f`.

Description: This command flushes the protocol addresses selected by some criteria.

Arguments: This command has the same arguments as `show`. The difference is that it does not run when no arguments are given.

Warning: This command (and other `flush` commands described below) is pretty dangerous. If you make a mistake, it will not forgive it, but will cruelly purge all the addresses.

Statistics: With the `-statistics` option, the command becomes verbose. It prints out the number of deleted addresses and the number of rounds made to flush the address list. If this option is given twice, `ip addr flush` also dumps all the deleted addresses in the format described in the previous subsection.

Example: Delete all the addresses from the private network 10.0.0.0/8:

```
netadm@amber:~ # ip -s -s a f to 10/8
2: dummy    inet 10.7.7.7/16 brd 10.7.255.255 scope global dummy
3: eth0    inet 10.10.7.7/16 brd 10.10.255.255 scope global eth0
4: eth1    inet 10.8.7.7/16 brd 10.8.255.255 scope global eth1

*** Round 1, deleting 3 addresses ***
*** Flush is complete after 1 round ***
netadm@amber:~ #
```

Another instructive example is disabling IP on all the Ethernets:

```
netadm@amber:~ # ip -4 addr flush label "eth*"
```

And the last example shows how to flush all the IPv6 addresses acquired by the host from stateless address autoconfiguration after you enabled forwarding or disabled autoconfiguration.

```
netadm@amber:~ # ip -6 addr flush dynamic
```

6 `ip neighbour` — neighbour/arp tables management

Abbreviations: `neighbour`, `neighbor`, `neigh`, `n`.

Object: `neighbour` objects establish bindings between protocol addresses and link layer addresses for hosts sharing the same link. Neighbour entries are organized into tables. The IPv4 neighbour table is known by another name — the ARP table.

The corresponding commands display neighbour bindings and their properties, add new neighbour entries and delete old ones.

Commands: `add`, `change`, `replace`, `delete`, `flush` and `show` (or `list`).

See also: Appendix B, p.49 describes how to manage proxy ARP/NDISC with the `ip` utility.

6.1 `ip neighbour add` — add a new neighbour entry `ip neighbour change` — change an existing entry `ip neighbour replace` — add a new entry or change an existing one

Abbreviations: `add`, `a`; `change`, `chg`; `replace`, `repl`.

Description: These commands create new neighbour records or update existing ones.

Arguments:

- `to ADDRESS` (default)

  — the protocol address of the neighbour. It is either an IPv4 or IPv6 address.

- `dev NAME`

  — the interface to which this neighbour is attached.

- `lladdr LLADDRESS`

  — the link layer address of the neighbour. `LLADDRESS` can also be `null`.

- `nud NUD_STATE`

  — the state of the neighbour entry. `nud` is an abbreviation for "Neighbour Unreachability Detection". The state can take one of the following values:

  1. `permanent` — the neighbour entry is valid forever and can be only be removed administratively.
  2. `noarp` — the neighbour entry is valid. No attempts to validate this entry will be made but it can be removed when its lifetime expires.
  3. `reachable` — the neighbour entry is valid until the reachability timeout expires.
  4. `stale` — the neighbour entry is valid but suspicious. This option to `ip neigh` does not change the neighbour state if it was valid and the address is not changed by this command.

Examples:

- `ip neigh add 10.0.0.3 lladdr 0:0:0:0:0:1 dev eth0 nud perm`

  — add a permanent ARP entry for the neighbour 10.0.0.3 on the device `eth0`.

- `ip neigh chg 10.0.0.3 dev eth0 nud reachable`

  — change its state to `reachable`.

6.2 `ip neighbour delete` — delete a neighbour entry

Abbreviations: `delete`, `del`, `d`.

Description: This command invalidates a neighbour entry.

Arguments: The arguments are the same as with `ip neigh add`, except that `lladdr` and `nud` are ignored.

Example:

- `ip neigh del 10.0.0.3 dev eth0`

  — invalidate an ARP entry for the neighbour 10.0.0.3 on the device `eth0`.

  NB. The deleted neighbour entry will not disappear from the tables immediately. If it is in use it cannot be deleted until the last client releases it. Otherwise it will be destroyed during the next garbage collection.

Warning: Attempts to delete or manually change a `noarp` entry created by the kernel may result in unpredictable behaviour. Particularly, the kernel may try to resolve this address even on a `NOARP` interface or if the address is multicast or broadcast.

6.3 `ip neighbour show` — list neighbour entries

Abbreviations: `show`, `list`, `sh`, `ls`.

Description: This commands displays neighbour tables.

Arguments:

- `to ADDRESS` (default)

  — the prefix selecting the neighbours to list.

- `dev NAME`

  — only list the neighbours attached to this device.

- `unused`

  — only list neighbours which are not currently in use.

- `nud NUD_STATE`

  — only list neighbour entries in this state. `NUD_STATE` takes values listed below or the special value `all` which means all states. This option may occur more than once. If this option is absent, `ip` lists all entries except for `none` and `noarp`.

Output format:

```
kuznet@alisa:~ $ ip neigh ls
:: dev lo lladdr 00:00:00:00:00:00 nud noarp
fe80::200:cff:fe76:3f85 dev eth0 lladdr 00:00:0c:76:3f:85 router \
    nud stale
0.0.0.0 dev lo lladdr 00:00:00:00:00:00 nud noarp
193.233.7.254 dev eth0 lladdr 00:00:0c:76:3f:85 nud reachable
193.233.7.85 dev eth0 lladdr 00:e0:1e:63:39:00 nud stale
kuznet@alisa:~ $
```

The first word of each line is the protocol address of the neighbour. Then the device name follows. The rest of the line describes the contents of the neighbour entry identified by the pair (device, address).

`lladdr` is the link layer address of the neighbour.

`nud` is the state of the "neighbour unreachability detection" machine for this entry. The detailed description of the neighbour state machine can be found in [1]. Here is the full list of the states with short descriptions:

1. `none` — the state of the neighbour is void.
2. `incomplete` — the neighbour is in the process of resolution.
3. `reachable` — the neighbour is valid and apparently reachable.
4. `stale` — the neighbour is valid, but is probably already unreachable, so the kernel will try to check it at the first transmission.
5. `delay` — a packet has been sent to the stale neighbour and the kernel is waiting for confirmation.
6. `probe` — the delay timer expired but no confirmation was received. The kernel has started to probe the neighbour with ARP/NDISC messages.
7. `failed` — resolution has failed.
8. `noarp` — the neighbour is valid. No attempts to check the entry will be made.
9. `permanent` — it is a `noarp` entry, but only the administrator may remove the entry from the neighbour table.

The link layer address is valid in all states except for `none`, `failed` and `incomplete`.

IPv6 neighbours can be marked with the additional flag `router` which means that the neighbour introduced itself as an IPv6 router [1].

Statistics: The `-statistics` option displays some usage statistics, f.e.

```
kuznet@alisa:~ $ ip -s n ls 193.233.7.254
193.233.7.254 dev eth0 lladdr 00:00:0c:76:3f:85 ref 5 used 12/13/20 \
    nud reachable
kuznet@alisa:~ $
```

Here `ref` is the number of users of this entry and `used` is a triplet of time intervals in seconds separated by slashes. In this case they show that:

1. the entry was used 12 seconds ago.
2. the entry was confirmed 13 seconds ago.
3. the entry was updated 20 seconds ago.

6.4 `ip neighbour flush` — flush neighbour entries

Abbreviations: `flush`, `f`.

Description: This command flushes neighbour tables, selecting entries to flush by some criteria.

Arguments: This command has the same arguments as `show`. The differences are that it does not run when no arguments are given, and that the default neighbour states to be flushed do not include `permanent` and `noarp`.

Statistics: With the `-statistics` option, the command becomes verbose. It prints out the number of deleted neighbours and the number of rounds made to flush the neighbour table. If the option is given twice, `ip neigh flush` also dumps all the deleted neighbours in the format described in the previous subsection.

Example:

```
netadm@alisa:~ # ip -s -s n f 193.233.7.254
193.233.7.254 dev eth0 lladdr 00:00:0c:76:3f:85 ref 5 used 12/13/20 \
    nud reachable

*** Round 1, deleting 1 entries ***
*** Flush is complete after 1 round ***
netadm@alisa:~ #
```

7 `ip route` — routing table management

Abbreviations: `route`, `ro`, `r`.

Object: `route` entries in the kernel routing tables keep information about paths to other networked nodes.

Each route entry has a *key* consisting of a *prefix* (i.e. a pair containing a network address and the length of its mask) and, optionally, the TOS value. An IP packet matches the route if the highest bits of its destination address are equal to the route prefix at least up to the prefix length and if the TOS of the route is zero or equal to the TOS of the packet.

If several routes match the packet, the following pruning rules are used to select the best one (see [3]):

1. The longest matching prefix is selected. All shorter ones are dropped.

2. If the TOS of some route with the longest prefix is equal to the TOS of the packet, the routes with different TOS are dropped.

   If no exact TOS match was found and routes with TOS=0 exist, the rest of routes are pruned.

   Otherwise, the route lookup fails.

3. If several routes remain after the previous steps, then the routes with the best preference values are selected.

4. If we still have several routes, then the *first* of them is selected.

   > NB. Note the ambiguity of the last step. Unfortunately, Linux historically allows such a bizarre situation. The sense of the word "first" depends on the order of route additions and it is practically impossible to maintain a bundle of such routes in this order.

   For simplicity we will limit ourselves to the case where such a situation is impossible and routes are uniquely identified by the triplet {prefix, tos, preference}. Actually, it is impossible to create non-unique routes with `ip` commands described in this section.

   One useful exception to this rule is the default route on non-forwarding hosts. It is "officially" allowed to have several fallback routes when several routers are present on directly connected networks. In this case, Linux-2.2 makes "dead gateway detection" [4] controlled by neighbour unreachability detection and by advice from transport protocols to select a working router, so the order of the routes is not essential. However, in this case, fiddling with default routes manually is not recommended. Use the Router Discovery protocol (see Appendix D, p.52) instead. Actually, Linux-2.2 IPv6 does not give user level applications any access to default routes.

Certainly, the steps above are not performed exactly in this sequence. Instead, the routing table in the kernel is kept in some data structure to achieve the final result with minimal cost. However, not depending on a particular routing algorithm implemented in the kernel, we can summarize the statements above as: a route is identified by the triplet {prefix, tos, preference}. This *key* lets us locate the route in the routing table.

Route attributes: Each route key refers to a routing information record containing the data required to deliver IP packets (f.e. output device and next hop router) and some optional attributes (f.e. the path MTU or the preferred source address when communicating with this destination). These attributes are described in the following subsection.

Route types: It is important that the set of required and optional attributes depend on the route *type*. The most important route type is `unicast`. It describes real paths to other hosts. As a rule, common routing tables contain only such routes. However, there are other types of routes with different semantics. The full list of types understood by Linux-2.2 is:

- `unicast` — the route entry describes real paths to the destinations covered by the route prefix.

- `unreachable` — these destinations are unreachable. Packets are discarded and the ICMP message *host unreachable* is generated. The local senders get an `EHOSTUNREACH` error.

- `blackhole` — these destinations are unreachable. Packets are discarded silently. The local senders get an `EINVAL` error.

- `prohibit` — these destinations are unreachable. Packets are discarded and the ICMP message *communication administratively prohibited* is generated. The local senders get an `EACCES` error.

- `local` — the destinations are assigned to this host. The packets are looped back and delivered locally.

- `broadcast` — the destinations are broadcast addresses. The packets are sent as link broadcasts.

- `throw` — a special control route used together with policy rules (see sec.8, p.35). If such a route is selected, lookup in this table is terminated pretending that no route was found. Without policy routing it is equivalent to the absence of the route in the routing table. The packets are dropped and the ICMP message *net unreachable* is generated. The local senders get an `ENETUNREACH` error.

- `nat` — a special NAT route. Destinations covered by the prefix are considered to be dummy (or external) addresses which require translation to real (or internal) ones before forwarding. The addresses to translate to are selected with the attribute `via`. More about NAT is in Appendix C, p.50.

- `anycast` — (*not implemented*) the destinations are *anycast* addresses assigned to this host. They are mainly equivalent to `local` with one difference: such addresses are invalid when used as the source address of any packet.

- `multicast` — a special type used for multicast routing. It is not present in normal routing tables.

Route tables: Linux-2.2 can pack routes into several routing tables identified by a number in the range from 1 to 255 or by name from the file `/etc/iproute2/rt_tables`. By default all normal routes are inserted into the `main` table (ID 254) and the kernel only uses this table when calculating routes.

Actually, one other table always exists, which is invisible but even more important. It is the `local` table (ID 255). This table consists of routes for local and broadcast addresses. The kernel maintains this table automatically and the administrator usually need not modify it or even look at it.

The multiple routing tables enter the game when *policy routing* is used. See sec.8, p.35. In this case, the table identifier effectively becomes one more parameter, which should be added to the triplet {prefix, tos, preference} to uniquely identify the route.

7.1 `ip route add` — add a new route
`ip route change` — change a route
`ip route replace` — change a route or add a new one

Abbreviations: `add`, `a`; `change`, `chg`; `replace`, `repl`.

Arguments:

- `to PREFIX` or `to TYPE PREFIX` (default)

  — the destination prefix of the route. If `TYPE` is omitted, `ip` assumes type `unicast`. Other values of `TYPE` are listed above. `PREFIX` is an IP or IPv6 address optionally followed by a slash and the prefix length. If the length of the prefix is missing, `ip` assumes a full-length host route. There is also a special `PREFIX` — `default` — which is equivalent to IP `0/0` or to IPv6 `::/0`.

- `tos TOS` or `dsfield TOS`

  — the Type Of Service (TOS) key. This key has no associated mask and the longest match is understood as: First, compare the TOS of the route and of the packet. If they are not equal, then the packet may still match a route with a zero TOS. `TOS` is either an 8 bit hexadecimal number or an identifier from `/etc/iproute2/rt_dsfield`.

- `metric NUMBER` or `preference NUMBER`

  — the preference value of the route. `NUMBER` is an arbitrary 32bit number.

- `table TABLEID`

  — the table to add this route to. `TABLEID` may be a number or a string from the file `/etc/iproute2/rt_tables`. If this parameter is omitted, `ip` assumes the `main` table, with the exception of `local`, `broadcast` and `nat` routes, which are put into the `local` table by default.

- `dev NAME`

  — the output device name.

- `via ADDRESS`

  — the address of the nexthop router. Actually, the sense of this field depends on the route type. For normal `unicast` routes it is either the true nexthop router or, if it is a direct route installed in BSD compatibility mode, it can be a local address of the interface. For NAT routes it is the first address of the block of translated IP destinations.

- `src ADDRESS`

  — the source address to prefer when sending to the destinations covered by the route prefix.

- `realm REALMID`

  — the realm to which this route is assigned. `REALMID` may be a number or a string from the file `/etc/iproute2/rt_realms`. Sec.13 (p.46) contains more information on realms.

- `mtu MTU` or `mtu lock MTU`

  — the MTU along the path to the destination. If the modifier `lock` is not used, the MTU may be updated by the kernel due to Path MTU Discovery. If the modifier `lock` is used, no path MTU discovery will be tried, all packets will be sent without the DF bit in IPv4 case or fragmented to MTU for IPv6.

- `window NUMBER`

  — the maximal window for TCP to advertise to these destinations, measured in bytes. It limits maximal data bursts that our TCP peers are allowed to send to us.

- `rtt NUMBER`

  — the initial RTT ("Round Trip Time") estimate.

- `rttvar NUMBER`

  — [2.3.15+ only] the initial RTT variance estimate.

- `ssthresh NUMBER`

  — [2.3.15+ only] an estimate for the initial slow start threshold.

- `cwnd NUMBER`

  — [2.3.15+ only] the clamp for congestion window. It is ignored if the `lock` flag is not used.

- `advmss NUMBER`

  — [2.3.15+ only] the MSS ("Maximal Segment Size") to advertise to these destinations when establishing TCP connections. If it is not given, Linux uses a default value calculated from the first hop device MTU.

  NB. If the path to these destination is asymmetric, this guess may be wrong.

- `reordering NUMBER`

  — [2.3.15+ only] Maximal reordering on the path to this destination. If it is not given, Linux uses the value selected with `sysctl` variable `net/ipv4/tcp_reordering`.

- `nexthop NEXTHOP`

  — the nexthop of a multipath route. `NEXTHOP` is a complex value with its own syntax similar to the top level argument lists:

  - `via ADDRESS` is the nexthop router.
  - `dev NAME` is the output device.
  - `weight NUMBER` is a weight for this element of a multipath route reflecting its relative bandwidth or quality.

- `scope SCOPE_VAL`

  — the scope of the destinations covered by the route prefix. `SCOPE_VAL` may be a number or a string from the file `/etc/iproute2/rt_scopes`. If this parameter is omitted, `ip` assumes scope `global` for all gatewayed `unicast` routes, scope `link` for direct `unicast` and `broadcast` routes and scope `host` for `local` routes.

- `protocol RTPROTO`

  — the routing protocol identifier of this route. `RTPROTO` may be a number or a string from the file `/etc/iproute2/rt_protos`. If the routing protocol ID is not given, `ip` assumes protocol `boot` (i.e. it assumes the route was added by someone who doesn't understand what they are doing). Several protocol values have a fixed interpretation. Namely:

- `redirect` — the route was installed due to an ICMP redirect.
- `kernel` — the route was installed by the kernel during autoconfiguration.
- `boot` — the route was installed during the bootup sequence. If a routing daemon starts, it will purge all of them.
- `static` — the route was installed by the administrator to override dynamic routing. Routing daemon will respect them and, probably, even advertise them to its peers.
- `ra` — the route was installed by Router Discovery protocol.

The rest of the values are not reserved and the administrator is free to assign (or not to assign) protocol tags. At least, routing daemons should take care of setting some unique protocol values, f.e. as they are assigned in `rtnetlink.h` or in `rt_protos` database.

- `onlink`

  — pretend that the nexthop is directly attached to this link, even if it does not match any interface prefix. One application of this option may be found in [6].

- `equalize`

  — allow packet by packet randomization on multipath routes. Without this modifier, the route will be frozen to one selected nexthop, so that load splitting will only occur on per-flow base. `equalize` only works if the kernel is patched.

NB. Actually there are more commands: `prepend` does the same thing as classic `route add`, i.e. adds a route, even if another route to the same destination exists. Its opposite case is `append`, which adds the route to the end of the list. Avoid these features.

NB. More sad news, IPv6 only understands the `append` command correctly. All the others are translated into `append` commands. Certainly, this will change in the future.

Examples:

- add a plain route to network 10.0.0/24 via gateway 193.233.7.65

```
ip route add 10.0.0/24 via 193.233.7.65
```

- change it to a direct route via the `dummy` device

```
ip ro chg 10.0.0/24 dev dummy
```

- add a default multipath route splitting the load between `ppp0` and `ppp1`

```
ip route add default scope global nexthop dev ppp0 \
                                  nexthop dev ppp1
```

  Note the scope value. It is not necessary but it informs the kernel that this route is gatewayed rather than direct. Actually, if you know the addresses of remote endpoints it would be better to use the `via` parameter.

- announce that the address 192.203.80.144 is not a real one, but should be translated to 193.233.7.83 before forwarding

```
ip route add nat 192.203.80.144 via 193.233.7.83
```

  Backward translation is setup with policy rules described in the following section (sec.8, p.35).

7.2 `ip route delete` — delete a route

Abbreviations: `delete`, `del`, `d`.

Arguments: `ip route del` has the same arguments as `ip route add`, but their semantics are a bit different.

Key values (`to`, `tos`, `preference` and `table`) select the route to delete. If optional attributes are present, `ip` verifies that they coincide with the attributes of the route to delete. If no route with the given key and attributes was found, `ip route del` fails.

NB. Linux-2.0 had the option to delete a route selected only by prefix address, ignoring its length (i.e. netmask). This option no longer exists because it was ambiguous. However, look at `ip route flush` (sec.7.4, p.31) which provides similar and even richer functionality.

Example:

- delete the multipath route created by the command in previous subsection

```
ip route del default scope global nexthop dev ppp0 \
                                  nexthop dev ppp1
```

7.3 `ip route show` — list routes

Abbreviations: `show`, `list`, `sh`, `ls`, `l`.

Description: the command displays the contents of the routing tables or the route(s) selected by some criteria.

Arguments:

- `to SELECTOR` (default)

  — only select routes from the given range of destinations. `SELECTOR` consists of an optional modifier (`root`, `match` or `exact`) and a prefix. `root PREFIX` selects routes with prefixes not shorter than `PREFIX`. F.e. `root 0/0` selects the entire routing table. `match PREFIX` selects routes with prefixes not longer than `PREFIX`. F.e. `match 10.0/16` selects `10.0/16`, `10/8` and `0/0`, but it does not select `10.1/16` and `10.0.0/24`. And `exact PREFIX` (or just `PREFIX`) selects routes with this exact prefix. If neither of these options are present, `ip` assumes `root 0/0` i.e. it lists the entire table.

- `tos TOS` or `dsfield TOS`

  — only select routes with the given TOS.

- `table TABLEID`

  — show the routes from this table(s). The default setting is to show `table main`. `TABLEID` may either be the ID of a real table or one of the special values:

  - `all` — list all of the tables.
  - `cache` — dump the routing cache.

  NB. IPv6 has a single table. However, splitting it into `main`, `local` and `cache` is emulated by the `ip` utility.

- `cloned` or `cached`

  — list cloned routes i.e. routes which were dynamically forked from other routes because some route attribute (f.e. MTU) was updated. Actually, it is equivalent to `table cache`.

- `from SELECTOR`

  — the same syntax as for `to`, but it binds the source address range rather than destinations. Note that the `from` option only works with cloned routes.

- `protocol RTPROTO`

  — only list routes of this protocol.

- `scope SCOPE_VAL`

  — only list routes with this scope.

- `type TYPE`

  — only list routes of this type.

- `dev NAME`

  — only list routes going via this device.

- `via PREFIX`

  — only list routes going via the nexthop routers selected by `PREFIX`.

- `src PREFIX`

  — only list routes with preferred source addresses selected by `PREFIX`.

- `realm REALMID` or `realms FROMREALM/TOREALM`

  — only list routes with these realms.

Examples: Let us count routes of protocol `gated/bgp` on a router:

```
kuznet@amber:~ $ ip ro ls proto gated/bgp | wc
   1413    9891    79010
kuznet@amber:~ $
```

To count the size of the routing cache, we have to use the `-o` option because cached attributes can take more than one line of output:

```
kuznet@amber:~ $ ip -o ro ls cloned | wc
   159    2543    18707
kuznet@amber:~ $
```

Output format: The output of this command consists of per route records separated by line feeds. However, some records may consist of more than one line: particularly, this is the case when the route is cloned or you requested additional statistics. If the `-o` option was given, then line feeds separating lines inside records are replaced with the backslash sign.

The output has the same syntax as arguments given to `ip route add`, so that it can be understood easily. F.e.

```
kuznet@amber:~ $ ip ro ls 193.233.7/24
193.233.7.0/24 dev eth0  proto gated/conn  scope link \
    src 193.233.7.65 realms inr.ac
kuznet@amber:~ $
```

If you list cloned entries, the output contains other attributes which are evaluated during route calculation and updated during route lifetime. An example of the output is:

```
kuznet@amber:~ $ ip ro ls 193.233.7.82 tab cache
193.233.7.82 from 193.233.7.82 dev eth0  src 193.233.7.65 \
  realms inr.ac/inr.ac
    cache <src-direct,redirect>  mtu 1500 rtt 300 iif eth0
193.233.7.82 dev eth0  src 193.233.7.65 realms inr.ac
    cache  mtu 1500 rtt 300
kuznet@amber:~ $
```

> NB. The route looks a bit strange, doesn't it? Did you notice that it is a path from 193.233.7.82 back to 193.233.82? Well, you will see in the section on `ip route get` (p.34) how it appeared.

The second line, starting with the word `cache`, shows additional attributes which normal routes do not possess. Cached flags are summarized in angle brackets:

- `local` — packets are delivered locally. It stands for loopback unicast routes, for broadcast routes and for multicast routes, if this host is a member of the corresponding group.
- `reject` — the path is bad. Any attempt to use it results in an error. See attribute `error` below (p.31).
- `mc` — the destination is multicast.
- `brd` — the destination is broadcast.
- `src-direct` — the source is on a directly connected interface.
- `redirected` — the route was created by an ICMP Redirect.
- `redirect` — packets going via this route will trigger an ICMP redirect.
- `fastroute` — the route is eligible to be used for fastroute.
- `equalize` — make packet by packet randomization along this path.
- `dst-nat` — the destination address requires translation.
- `src-nat` — the source address requires translation.
- `masq` — the source address requires masquerading. This feature disappeared in linux-2.4.
- `notify` — (*not implemented*) change/deletion of this route will trigger RTNETLINK notification.

Then some optional attributes follow:

- `error` — on `reject` routes it is error code returned to local senders when they try to use this route. These error codes are translated into ICMP error codes, sent to remote senders, according to the rules described above in the subsection devoted to route types (p.22).
- `expires` — this entry will expire after this timeout.
- `iif` — the packets for this path are expected to arrive on this interface.

Statistics: With the `-statistics` option, more information about this route is shown:

- `users` — the number of users of this entry.
- `age` — shows when this route was last used.
- `used` — the number of lookups of this route since its creation.

7.4 `ip route flush` — flush routing tables

Abbreviations: `flush`, `f`.

Description: this command flushes routes selected by some criteria.

Arguments: the arguments have the same syntax and semantics as the arguments of `ip route show`, but routing tables are not listed but purged. The only difference is the default action: `show` dumps all the IP main routing table but `flush` prints the helper page. The reason for this difference does not require any explanation, does it?

Statistics: With the `-statistics` option, the command becomes verbose. It prints out the number of deleted routes and the number of rounds made to flush the routing table. If the option is given twice, `ip route flush` also dumps all the deleted routes in the format described in the previous subsection.

Examples: The first example flushes all the gatewayed routes from the main table (f.e. after a routing daemon crash).

```
netadm@amber:~ # ip -4 ro flush scope global type unicast
```

This option deserves to be put into a scriptlet `routef`.

NB. This option was described in the `route(8)` man page borrowed from BSD, but was never implemented in Linux.

The second example flushes all IPv6 cloned routes:

```
netadm@amber:~ # ip -6 -s -s ro flush cache
3ffe:2400::220:afff:fef4:c5d1 via 3ffe:2400::220:afff:fef4:c5d1 \
  dev eth0  metric 0
    cache  used 2 age 12sec mtu 1500 rtt 300
3ffe:2400::280:adff:feb7:8034 via 3ffe:2400::280:adff:feb7:8034 \
  dev eth0  metric 0
    cache  used 2 age 15sec mtu 1500 rtt 300
3ffe:2400::280:c8ff:fe59:5bcc via 3ffe:2400::280:c8ff:fe59:5bcc \
  dev eth0  metric 0
    cache  users 1 used 1 age 23sec mtu 1500 rtt 300
3ffe:2400:0:1:2a0:ccff:fe66:1878 via 3ffe:2400:0:1:2a0:ccff:fe66:1878 \
  dev eth1  metric 0
    cache  used 2 age 20sec mtu 1500 rtt 300
3ffe:2400:0:1:a00:20ff:fe71:fb30 via 3ffe:2400:0:1:a00:20ff:fe71:fb30 \
  dev eth1  metric 0
    cache  used 2 age 33sec mtu 1500 rtt 300
ff02::1 via ff02::1 dev eth1  metric 0
    cache  users 1 used 1 age 45sec mtu 1500 rtt 300

*** Round 1, deleting 6 entries ***
*** Flush is complete after 1 round ***
netadm@amber:~ # ip -6 -s -s ro flush cache
Nothing to flush.
netadm@amber:~ #
```

The third example flushes BGP routing tables after a `gated` death.

```
netadm@amber:~ # ip ro ls proto gated/bgp | wc
   1408    9856    78730
netadm@amber:~ # ip -s ro f proto gated/bgp

*** Round 1, deleting 1408 entries ***
*** Flush is complete after 1 round ***
netadm@amber:~ # ip ro f proto gated/bgp
Nothing to flush.
netadm@amber:~ # ip ro ls proto gated/bgp
netadm@amber:~ #
```

7.5 `ip route get` — get a single route

Abbreviations: `get`, `g`.

Description: this command gets a single route to a destination and prints its contents exactly as the kernel sees it.

Arguments:

- `to ADDRESS` (default)

  — the destination address.

- `from ADDRESS`

  — the source address.

- `tos TOS` or `dsfield TOS`

  — the Type Of Service.

- `iif NAME`

  — the device from which this packet is expected to arrive.

- `oif NAME`

  — force the output device on which this packet will be routed.

- `connected`

  — if no source address (option `from`) was given, relookup the route with the source set to the preferred address received from the first lookup. If policy routing is used, it may be a different route.

Note that this operation is not equivalent to `ip route show`. `show` shows existing routes. `get` resolves them and creates new clones if necessary. Essentially, `get` is equivalent to sending a packet along this path. If the `iif` argument is not given, the kernel creates a route to output packets towards the requested destination. This is equivalent to pinging the destination with a subsequent `ip route ls cache`, however, no packets are actually sent. With the `iif` argument, the kernel pretends that a packet arrived from this interface and searches for a path to forward the packet.

Output format: This command outputs routes in the same format as `ip route ls`.

Examples:

- Find a route to output packets to 193.233.7.82:

```
kuznet@amber:~ $ ip route get 193.233.7.82
193.233.7.82 dev eth0  src 193.233.7.65 realms inr.ac
    cache  mtu 1500 rtt 300
kuznet@amber:~ $
```

- Find a route to forward packets arriving on `eth0` from 193.233.7.82 and destined for 193.233.7.82:

```
kuznet@amber:~ $ ip r g 193.233.7.82 from 193.233.7.82 iif eth0
193.233.7.82 from 193.233.7.82 dev eth0  src 193.233.7.65 \
  realms inr.ac/inr.ac
    cache <src-direct,redirect>  mtu 1500 rtt 300 iif eth0
kuznet@amber:~ $
```

NB. This is the command that created the funny route from 193.233.7.82 looped back to 193.233.7.82 (cf. NB on p.30). Note the `redirect` flag on it.

- Find a multicast route for packets arriving on `eth0` from host 193.233.7.82 and destined for multicast group 224.2.127.254 (it is assumed that a multicast routing daemon is running. In this case, it is `pimd`)

```
kuznet@amber:~ $ ip r g 224.2.127.254 from 193.233.7.82 iif eth0
multicast 224.2.127.254 from 193.233.7.82 dev lo  \
  src 193.233.7.65 realms inr.ac/cosmos
    cache <mc> iif eth0 Oifs: eth1 pimreg
kuznet@amber:~ $
```

  This route differs from the ones seen before. It contains a "normal" part and a "multicast" part. The normal part is used to deliver (or not to deliver) the packet to local IP listeners. In this case the router is not a member of this group, so that route has no `local` flag and only forwards packets. The output device for such entries is always loopback. The multicast part consists of an additional `Oifs:` list showing the output interfaces.

It is time for a more complicated example. Let us add an invalid gatewayed route for a destination which is really directly connected:

```
netadm@alisa:~ # ip route add 193.233.7.98 via 193.233.7.254
netadm@alisa:~ # ip route get 193.233.7.98
193.233.7.98 via 193.233.7.254 dev eth0  src 193.233.7.90
    cache  mtu 1500 rtt 3072
netadm@alisa:~ #
```

and probe it with ping:

```
netadm@alisa:~ # ping -n 193.233.7.98
PING 193.233.7.98 (193.233.7.98) from 193.233.7.90 : 56 data bytes
From 193.233.7.254: Redirect Host(New nexthop: 193.233.7.98)
64 bytes from 193.233.7.98: icmp_seq=0 ttl=255 time=3.5 ms
From 193.233.7.254: Redirect Host(New nexthop: 193.233.7.98)
64 bytes from 193.233.7.98: icmp_seq=1 ttl=255 time=2.2 ms
64 bytes from 193.233.7.98: icmp_seq=2 ttl=255 time=0.4 ms
```

```
64 bytes from 193.233.7.98: icmp_seq=3 ttl=255 time=0.4 ms
64 bytes from 193.233.7.98: icmp_seq=4 ttl=255 time=0.4 ms
^C
--- 193.233.7.98 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max = 0.4/1.3/3.5 ms
netadm@alisa:~ #
```

What happened? Router 193.233.7.254 understood that we have a much better path to the destination and sent us an ICMP redirect message. We may retry `ip route get` to see what we have in the routing tables now:

```
netadm@alisa:~ # ip route get 193.233.7.98
193.233.7.98 dev eth0  src 193.233.7.90
    cache <redirected>  mtu 1500 rtt 3072
netadm@alisa:~ #
```

8 `ip rule` — routing policy database management

Abbreviations: `rule`, `ru`.

Object: `rule`s in the routing policy database control the route selection algorithm.

Classic routing algorithms used in the Internet make routing decisions based only on the destination address of packets (and in theory, but not in practice, on the TOS field). The seminal review of classic routing algorithms and their modifications can be found in [3].

In some circumstances we want to route packets differently depending not only on destination addresses, but also on other packet fields: source address, IP protocol, transport protocol ports or even packet payload. This task is called "policy routing".

NB. "policy routing" $\neq$ "routing policy".
"policy routing" = "cunning routing".
"routing policy" = "routing tactics" or "routing plan".

To solve this task, the conventional destination based routing table, ordered according to the longest match rule, is replaced with a "routing policy database" (or RPDB), which selects routes by executing some set of rules. The rules may have lots of keys of different natures and therefore they have no natural ordering, but one imposed by the administrator. Linux-2.2 RPDB is a linear list of rules ordered by numeric priority value. RPDB explicitly allows matching a few packet fields:

- packet source address.
- packet destination address.

- TOS.
- incoming interface (which is packet metadata, rather than a packet field).

Matching IP protocols and transport ports is also possible, indirectly, via `ipchains`, by exploiting their ability to mark some classes of packets with `fwmark`. Therefore, `fwmark` is also included in the set of keys checked by rules.

Each policy routing rule consists of a *selector* and an *action* predicate. The RPDB is scanned in the order of increasing priority. The selector of each rule is applied to {source address, destination address, incoming interface, tos, fwmark} and, if the selector matches the packet, the action is performed. The action predicate may return with success. In this case, it will either give a route or failure indication and the RPDB lookup is terminated. Otherwise, the RPDB program continues on the next rule.

What is the action, semantically? The natural action is to select the nexthop and the output device. This is what Cisco IOS [5] does. Let us call it "match & set". The Linux-2.2 approach is more flexible. The action includes lookups in destination-based routing tables and selecting a route from these tables according to the classic longest match algorithm. The "match & set" approach is the simplest case of the Linux one. It is realized when a second level routing table contains a single default route. Recall that Linux-2.2 supports multiple tables managed with the `ip route` command, described in the previous section.

At startup time the kernel configures the default RPDB consisting of three rules:

1. Priority: 0, Selector: match anything, Action: lookup routing table `local` (ID 255). The `local` table is a special routing table containing high priority control routes for local and broadcast addresses.

   Rule 0 is special. It cannot be deleted or overridden.

2. Priority: 32766, Selector: match anything, Action: lookup routing table `main` (ID 254). The `main` table is the normal routing table containing all non-policy routes. This rule may be deleted and/or overridden with other ones by the administrator.

3. Priority: 32767, Selector: match anything, Action: lookup routing table `default` (ID 253). The `default` table is empty. It is reserved for some post-processing if no previous default rules selected the packet. This rule may also be deleted.

Do not confuse routing tables with rules: rules point to routing tables, several rules may refer to one routing table and some routing tables may have no rules pointing to them. If the administrator deletes all the rules referring to a table, the table is not used, but it still exists and will disappear only after all the routes contained in it are deleted.

Rule attributes: Each RPDB entry has additional attributes. F.e. each rule has a pointer to some routing table. NAT and masquerading rules have an attribute to select new IP address to translate/masquerade. Besides that, rules have some optional attributes, which routes have, namely `realms`. These values do not override those contained in the routing tables. They are only used if the route did not select any attributes.

Rule types: The RPDB may contain rules of the following types:

- `unicast` — the rule prescribes to return the route found in the routing table referenced by the rule.
- `blackhole` — the rule prescribes to silently drop the packet.
- `unreachable` — the rule prescribes to generate a "Network is unreachable" error.
- `prohibit` — the rule prescribes to generate "Communication is administratively prohibited" error.
- `nat` — the rule prescribes to translate the source address of the IP packet into some other value. More about NAT is in Appendix C, p.50.

Commands: `add`, `delete` and `show` (or `list`).

8.1 `ip rule add` — insert a new rule
`ip rule delete` — delete a rule

Abbreviations: `add`, `a`; `delete`, `del`, `d`.

Arguments:

- `type TYPE` (default)

  — the type of this rule. The list of valid types was given in the previous subsection.

- `from PREFIX`

  — select the source prefix to match.

- `to PREFIX`

  — select the destination prefix to match.

- `iif NAME`

  — select the incoming device to match. If the interface is loopback, the rule only matches packets originating from this host. This means that you may create separate routing tables for forwarded and local packets and, hence, completely segregate them.

- `tos TOS` or `dsfield TOS`

  — select the TOS value to match.

- `fwmark MARK`

  — select the `fwmark` value to match.

- `priority PREFERENCE`

  — the priority of this rule. Each rule should have an explicitly set *unique* priority value.

  > NB. Really, for historical reasons `ip rule add` does not require a priority value and allows them to be non-unique. If the user does not supplied a priority, it is selected by the kernel. If the user creates a rule with a priority value that already exists, the kernel does not reject the request. It adds the new rule before all old rules of the same priority.
  >
  > It is mistake in design, no more. And it will be fixed one day, so do not rely on this feature. Use explicit priorities.

- `table TABLEID`

  — the routing table identifier to lookup if the rule selector matches.

- `realms FROM/TO`

  — Realms to select if the rule matched and the routing table lookup succeeded. Realm `TO` is only used if the route did not select any realm.

- `nat ADDRESS`

  — The base of the IP address block to translate (for source addresses). The `ADDRESS` may be either the start of the block of NAT addresses (selected by NAT routes) or in linux-2.2 a local host address (or even zero). In the last case the router does not translate the packets, but masquerades them to this address; this feature disappered in 2.4. More about NAT is in Appendix C, p.50.

Warning: Changes to the RPDB made with these commands do not become active immediately. It is assumed that after a script finishes a batch of updates, it flushes the routing cache with `ip route flush cache`.

Examples:

- Route packets with source addresses from 192.203.80/24 according to routing table `inr.ruhep`:

  ```
  ip ru add from 192.203.80.0/24 table inr.ruhep prio 220
  ```

- Translate packet source address 193.233.7.83 into 192.203.80.144 and route it according to table #1 (actually, it is `inr.ruhep`):

  ```
  ip ru add from 193.233.7.83 nat 192.203.80.144 table 1 prio 320
  ```

- Delete the unused default rule:

  ```
  ip ru del prio 32767
  ```

8.2 `ip rule show` — list rules

Abbreviations: `show`, `list`, `sh`, `ls`, `l`.

Arguments: Good news, this is one command that has no arguments.

Output format:

```
kuznet@amber:~ $ ip ru ls
0: from all lookup local
200: from 192.203.80.0/24 to 193.233.7.0/24 lookup main
210: from 192.203.80.0/24 to 192.203.80.0/24 lookup main
220: from 192.203.80.0/24 lookup inr.ruhep realms inr.ruhep/radio-msu
300: from 193.233.7.83 to 193.233.7.0/24 lookup main
310: from 193.233.7.83 to 192.203.80.0/24 lookup main
320: from 193.233.7.83 lookup inr.ruhep map-to 192.203.80.144
32766: from all lookup main
kuznet@amber:~ $
```

In the first column is the rule priority value followed by a colon. Then the selectors follow. Each key is prefixed with the same keyword that was used to create the rule.

The keyword `lookup` is followed by a routing table identifier, as it is recorded in the file `/etc/iproute2/rt_tables`.

If the rule does NAT (f.e. rule #320), it is shown by the keyword `map-to` followed by the start of the block of addresses to map.

The sense of this example is pretty simple. The prefixes 192.203.80.0/24 and 193.233.7.0/24 form the internal network, but they are routed differently when the packets leave it. Besides that, the host 193.233.7.83 is translated into another prefix to look like 192.203.80.144 when talking to the outer world.

9 `ip maddress` — multicast addresses management

Object: `maddress` objects are multicast addresses.

Commands: `add`, `delete`, `show` (or `list`).

9.1 `ip maddress show` — list multicast addresses

Abbreviations: `show`, `list`, `sh`, `ls`, `l`.

Arguments:

- `dev NAME` (default)

  — the device name.

Output format:

```
kuznet@alisa:~ $ ip maddr ls dummy
2:  dummy
    link  33:33:00:00:00:01
    link  01:00:5e:00:00:01
    inet  224.0.0.1 users 2
    inet6 ff02::1
kuznet@alisa:~ $
```

The first line of the output shows the interface index and its name. Then the multicast address list follows. Each line starts with the protocol identifier. The word `link` denotes a link layer multicast addresses.

If a multicast address has more than one user, the number of users is shown after the `users` keyword.

One additional feature not present in the example above is the `static` flag, which indicates that the address was joined with `ip maddr add`. See the following subsection.

9.2 `ip maddress add` — add a multicast address `ip maddress delete` — delete a multicast address

Abbreviations: `add`, `a`; `delete`, `del`, `d`.

Description: these commands attach/detach a static link layer multicast address to listen on the interface. Note that it is impossible to join protocol multicast groups statically. This command only manages link layer addresses.

Arguments:

- `address LLADDRESS` (default)
  — the link layer multicast address.
- `dev NAME`
  — the device to join/leave this multicast address.

Example: Let us continue with the example from the previous subsection.

```
netadm@alisa:~ # ip maddr add 33:33:00:00:00:01 dev dummy
netadm@alisa:~ # ip -O maddr ls dummy
2:  dummy
    link  33:33:00:00:00:01 users 2 static
    link  01:00:5e:00:00:01
netadm@alisa:~ # ip maddr del 33:33:00:00:00:01 dev dummy
```

> NB. Neither `ip` nor the kernel check for multicast address validity. Particularly, this means that you can try to load a unicast address instead of a multicast address. Most drivers will ignore such addresses, but several (f.e. Tulip) will intern it to their on-board filter. The effects may be strange. Namely, the addresses become additional local link addresses and, if you loaded the address of another host to the router, wait for duplicated packets on the wire. It is not a bug, but rather a hole in the API and intra-kernel interfaces. This feature is really more useful for traffic monitoring, but using it with Linux-2.2 you *have to* be sure that the host is not a router and, especially, that it is not a transparent proxy or masquerading agent.

10 `ip mroute` — multicast routing cache management

Abbreviations: `mroute`, `mr`.

Object: `mroute` objects are multicast routing cache entries created by a user level mrouting daemon (f.e. `pimd` or `mrouted`).

Due to the limitations of the current interface to the multicast routing engine, it is impossible to change `mroute` objects administratively, so we may only display them. This limitation will be removed in the future.

Commands: `show` (or `list`).

10.1 `ip mroute show` — list mroute cache entries

Abbreviations: `show`, `list`, `sh`, `ls`, `l`.

Arguments:

- `to PREFIX` (default)

  — the prefix selecting the destination multicast addresses to list.

- `iif NAME`

  — the interface on which multicast packets are received.

- `from PREFIX`

  — the prefix selecting the IP source addresses of the multicast route.

Output format:

```
kuznet@amber:~ $ ip mroute ls
(193.232.127.6, 224.0.1.39)      Iif: unresolved
(193.232.244.34, 224.0.1.40)     Iif: unresolved
(193.233.7.65, 224.66.66.66)     Iif: eth0       Oifs: pimreg
kuznet@amber:~ $
```

Each line shows one (S,G) entry in the multicast routing cache, where S is the source address and G is the multicast group. `Iif` is the interface on which multicast packets are expected to arrive. If the word `unresolved` is there instead of the interface name, it means that the routing daemon still hasn't resolved this entry. The keyword `oifs` is followed by a list of output interfaces, separated by spaces. If a multicast routing entry is created with non-trivial TTL scope, administrative distances are appended to the device names in the `oifs` list.

Statistics: The `-statistics` option also prints the number of packets and bytes forwarded along this route and the number of packets that arrived on the wrong interface, if this number is not zero.

```
kuznet@amber:~ $ ip -s mr ls 224.66/16
(193.233.7.65, 224.66.66.66)     Iif: eth0       Oifs: pimreg
  9383 packets, 300256 bytes
kuznet@amber:~ $
```

11 `ip tunnel` — tunnel configuration

Abbreviations: `tunnel`, `tunl`.

Object: `tunnel` objects are tunnels, encapsulating packets in IPv4 packets and then sending them over the IP infrastructure.

Commands: `add`, `delete`, `change`, `show` (or `list`).

See also: A more informal discussion of tunneling over IP and the `ip tunnel` command can be found in [6].

11.1 `ip tunnel add` — add a new tunnel `ip tunnel change` — change an existing tunnel `ip tunnel delete` — destroy a tunnel

Abbreviations: `add`, `a`; `change`, `chg`; `delete`, `del`, `d`.

Arguments:

- `name NAME` (default)

  — select the tunnel device name.

- `mode MODE`

  — set the tunnel mode. Three modes are currently available: `ipip`, `sit` and `gre`.

- `remote ADDRESS`

  — set the remote endpoint of the tunnel.

- `local ADDRESS`

  — set the fixed local address for tunneled packets. It must be an address on another interface of this host.

- `ttl N`

  — set a fixed TTL `N` on tunneled packets. `N` is a number in the range 1–255. 0 is a special value meaning that packets inherit the TTL value. The default value is: `inherit`.

- `tos T` or `dsfield T`

  — set a fixed TOS `T` on tunneled packets. The default value is: `inherit`.

- `dev NAME`

  — bind the tunnel to the device `NAME` so that tunneled packets will only be routed via this device and will not be able to escape to another device when the route to endpoint changes.

- `nopmtudisc`

  — disable Path MTU Discovery on this tunnel. It is enabled by default. Note that a fixed ttl is incompatible with this option: tunnelling with a fixed ttl always makes pmtu discovery.

- `key K`, `ikey K`, `okey K`

  — (only GRE tunnels) use keyed GRE with key `K`. `K` is either a number or an IP address-like dotted quad. The `key` parameter sets the key to use in both directions. The `ikey` and `okey` parameters set different keys for input and output.

- `csum`, `icsum`, `ocsum`

  — (only GRE tunnels) generate/require checksums for tunneled packets. The `ocsum` flag calculates checksums for outgoing packets. The `icsum` flag requires that all input packets have the correct checksum. The `csum` flag is equivalent to the combination "`icsum ocsum`".

- `seq`, `iseq`, `oseq`

  — (only GRE tunnels) serialize packets. The `oseq` flag enables sequencing of outgoing packets. The `iseq` flag requires that all input packets are serialized. The `seq` flag is equivalent to the combination "`iseq oseq`".

  > NB. I think this option does not work. At least, I did not test it, did not debug it and do not even understand how it is supposed to work or for what purpose Cisco planned to use it. Do not use it.

Example: Create a pointopoint IPv6 tunnel with maximal TTL of 32.

```
netadm@amber:~ # ip tunl add Cisco mode sit remote 192.31.7.104 \
    local 192.203.80.142 ttl 32
```

11.2 `ip tunnel show` — list tunnels

Abbreviations: `show`, `list`, `sh`, `ls`, `l`.

Arguments: None.

Output format:

```
kuznet@amber:~ $ ip tunl ls Cisco
Cisco: ipv6/ip  remote 192.31.7.104  local 192.203.80.142  ttl 32
kuznet@amber:~ $
```

The line starts with the tunnel device name followed by a colon. Then the tunnel mode follows. The parameters of the tunnel are listed with the same keywords that were used when creating the tunnel.

Statistics:

```
kuznet@amber:~ $ ip -s tunl ls Cisco
Cisco: ipv6/ip  remote 192.31.7.104  local 192.203.80.142  ttl 32
RX: Packets    Bytes        Errors CsumErrs OutOfSeq Mcasts
    12566      1707516      0      0        0        0
TX: Packets    Bytes        Errors DeadLoop NoRoute  NoBufs
    13445      1879677      0      0        0        0
kuznet@amber:~ $
```

Essentially, these numbers are the same as the numbers printed with `ip -s link show` (sec.4.2, p.7) but the tags are different to reflect that they are tunnel specific.

- `CsumErrs` — the total number of packets dropped because of checksum failures for a GRE tunnel with checksumming enabled.
- `OutOfSeq` — the total number of packets dropped because they arrived out of sequence for a GRE tunnel with serialization enabled.
- `Mcasts` — the total number of multicast packets received on a broadcast GRE tunnel.
- `DeadLoop` — the total number of packets which were not transmitted because the tunnel is looped back to itself.
- `NoRoute` — the total number of packets which were not transmitted because there is no IP route to the remote endpoint.
- `NoBufs` — the total number of packets which were not transmitted because the kernel failed to allocate a buffer.

12 `ip monitor` and `rtmon` — state monitoring

The `ip` utility can monitor the state of devices, addresses and routes continuously. This option has a slightly different format. Namely, the `monitor` command is the first in the command line and then the object list follows:

```
ip monitor [ file FILE ] [ all | OBJECT-LIST ]
```

`OBJECT-LIST` is the list of object types that we want to monitor. It may contain `link`, `address` and `route`. If no `file` argument is given, `ip` opens RTNETLINK, listens on it and dumps state changes in the format described in previous sections.

If a file name is given, it does not listen on RTNETLINK, but opens the file containing RTNETLINK messages saved in binary format and dumps them. Such a history file can be generated with the `rtmon` utility. This utility has a command line syntax similar to `ip monitor`. Ideally, `rtmon` should be started before the first network configuration command is issued. F.e. if you insert:

```
rtmon file /var/log/rtmon.log
```

in a startup script, you will be able to view the full history later.

Certainly, it is possible to start `rtmon` at any time. It prepends the history with the state snapshot dumped at the moment of starting.

13 Route realms and policy propagation, `rtacct`

On routers using OSPF ASE or, especially, the BGP protocol, routing tables may be huge. If we want to classify or to account for the packets per route, we will have to keep lots of information. Even worse, if we want to distinguish the packets not only by their destination, but also by their source, the task gets quadratic complexity and its solution is physically impossible.

One approach to propagating the policy from routing protocols to the forwarding engine has been proposed in [8]. Essentially, Cisco Policy Propagation via BGP is based on the fact that dedicated routers all have the RIB (Routing Information Base) close to the forwarding engine, so policy routing rules can check all the route attributes, including ASPATH information and community strings.

The Linux architecture, splitting the RIB (maintained by a user level daemon) and the kernel based FIB (Forwarding Information Base), does not allow such a simple approach.

It is to our fortune because there is another solution which allows even more flexible policy and richer semantics.

Namely, routes can be clustered together in user space, based on their attributes. F.e. a BGP router knows route ASPATH, its community; an OSPF router knows the route tag or its area. The administrator, when adding routes manually, also knows their nature. Providing that the number of such aggregates (we call them *realms*) is low, the task of full classification both by source and destination becomes quite manageable.

So each route may be assigned to a realm. It is assumed that this identification is made by a routing daemon, but static routes can also be handled manually with `ip route` (see sec.7, p.21).

> NB. There is a patch to `gated`, allowing classification of routes to realms with all the set of policy rules implemented in `gated`: by prefix, by ASPATH, by origin, by tag etc.

To facilitate the construction (f.e. in case the routing daemon is not aware of realms), missing realms may be completed with routing policy rules, see sec. 8, p.35.

For each packet the kernel calculates a tuple of realms: source realm and destination realm, using the following algorithm:

1. If the route has a realm, the destination realm of the packet is set to it.

2. If the rule has a source realm, the source realm of the packet is set to it. If the destination realm was not inherited from the route and the rule has a destination realm, it is also set.

3. If at least one of the realms is still unknown, the kernel finds the reversed route to the source of the packet.

4. If the source realm is still unknown, get it from the reversed route.

5. If one of the realms is still unknown, swap the realms of reversed routes and apply step 2 again.

After this procedure is completed we know what realm the packet arrived from and the realm where it is going to propagate to. If some of the realms are unknown, they are initialized to zero (or realm `unknown`).

The main application of realms is the TC `route` classifier [7], where they are used to help assign packets to traffic classes, to account, police and schedule them according to this classification.

A much simpler but still very useful application is incoming packet accounting by realms. The kernel gathers a packet statistics summary which can be viewed with the `rtacct` utility.

```
kuznet@amber:~ $ rtacct russia
Realm       BytesTo     PktsTo      BytesFrom   PktsFrom
russia      20576778    169176      47080168    153805
kuznet@amber:~ $
```

This shows that this router received 153805 packets from the realm `russia` and forwarded 169176 packets to `russia`. The realm `russia` consists of routes with ASPATHs not leaving Russia.

Note that locally originating packets are not accounted here, `rtacct` shows incoming packets only. Using the `route` classifier (see [7]) you can get even more detailed accounting information about outgoing packets, optionally summarizing traffic not only by source or destination, but by any pair of source and destination realms.

References

[1] T. Narten, E. Nordmark, W. Simpson. "Neighbor Discovery for IP Version 6 (IPv6)", RFC-2461.

[2] S. Thomson, T. Narten. "IPv6 Stateless Address Autoconfiguration", RFC-2462.

[3] F. Baker. "Requirements for IP Version 4 Routers", RFC-1812.

[4] R. T. Braden. "Requirements for Internet hosts — communication layers", RFC-1122.

[5] "Cisco IOS Release 12.0 Network Protocols Command Reference, Part 1" and "Cisco IOS Release 12.0 Quality of Service Solutions Configuration Guide: Configuring Policy-Based Routing", http://www.cisco.com/univercd/cc/td/doc/product/software/ios120.

[6] A. N. Kuznetsov. "Tunnels over IP in Linux-2.2", In: `ftp://ftp.inr.ac.ru/ip-routing/iproute2-current.tar.gz`.

[7] A. N. Kuznetsov. "TC Command Reference", In: `ftp://ftp.inr.ac.ru/ip-routing/iproute2-current.tar.gz`.

[8] "Cisco IOS Release 12.0 Quality of Service Solutions Configuration Guide: Configuring QoS Policy Propagation via Border Gateway Protocol", http://www.cisco.com/univercd/cc/td/doc/product/software/ios120.

[9] R. Droms. "Dynamic Host Configuration Protocol.", RFC-2131

A Source address selection

When a host creates an IP packet, it must select some source address. Correct source address selection is a critical procedure, because it gives the receiver the information needed to deliver a reply. If the source is selected incorrectly, in the best case, the backward path may appear different to the forward one which is harmful for performance. In the worst case, when the addresses are administratively scoped, the reply may be lost entirely.

Linux-2.2 selects source addresses using the following algorithm:

- The application may select a source address explicitly with `bind(2)` syscall or supplying it to `sendmsg(2)` via the ancillary data object `IP_PKTINFO`. In this case the kernel only checks the validity of the address and never tries to "improve" an incorrect user choice, generating an error instead.

NB. Never say “Never”. The sysctl option `ip_dynaddr` breaks this axiom. It has been made deliberately with the purpose of automatically reselecting the address on hosts with dynamic dial-out interfaces. However, this hack *must not* be used on multihomed hosts and especially on routers: it would break them.

- Otherwise, IP routing tables can contain an explicit source address hint for this destination. The hint is set with the `src` parameter to the `ip route` command, sec.7, p.21.
- Otherwise, the kernel searches through the list of addresses attached to the interface through which the packets will be routed. The search strategies are different for IP and IPv6. Namely:
  - IPv6 searches for the first valid, not deprecated address with the same scope as the destination.
  - IP searches for the first valid address with a scope wider than the scope of the destination but it prefers addresses which fall to the same subnet as the nexthop of the route to the destination. Unlike IPv6, the scopes of IPv4 destinations are not encoded in their addresses but are supplied in routing tables instead (the `scope` parameter to the `ip route` command, sec.7, p.21).
- Otherwise, if the scope of the destination is `link` or `host`, the algorithm fails and returns a zero source address.
- Otherwise, all interfaces are scanned to search for an address with an appropriate scope. The loopback device `lo` is always the first in the search list, so that if an address with global scope (not 127.0.0.1!) is configured on loopback, it is always preferred.

B Proxy ARP/NDISC

Routers may answer ARP/NDISC solicitations on behalf of other hosts. In Linux-2.2 proxy ARP on an interface may be enabled by setting the kernel `sysctl` variable `/proc/sys/net/ipv4/conf/<dev>/proxy_arp` to 1. After this, the router starts to answer ARP requests on the interface `<dev>`, provided the route to the requested destination does *not* go back via the same device.

The variable `/proc/sys/net/ipv4/conf/all/proxy_arp` enables proxy ARP on all the IP devices.

However, this approach fails in the case of IPv6 because the router must join the solicited node multicast address to listen for the corresponding NDISC queries. It means that proxy NDISC is possible only on a per destination basis.

Logically, proxy ARP/NDISC is not a kernel task. It can easily be implemented in user space. However, similar functionality was present in BSD kernels and in Linux-2.0, so we have to preserve it at least to the extent that is standardized in BSD.

> NB. Linux-2.0 ARP had a feature called *subnet* proxy ARP. It is replaced with the sysctl flag in Linux-2.2.

The `ip` utility provides a way to manage proxy ARP/NDISC with the `ip neigh` command, namely:

```
ip neigh add proxy ADDRESS [ dev NAME ]
```

adds a new proxy ARP/NDISC record and

```
ip neigh del proxy ADDRESS [ dev NAME ]
```

deletes it.

If the name of the device is not given, the router will answer solicitations for address `ADDRESS` on all devices, otherwise it will only serve the device `NAME`. Even if the proxy entry is created with `ip neigh`, the router *will not* answer a query if the route to the destination goes back via the interface from which the solicitation was received.

It is important to emphasize that proxy entries have *no* parameters other than these (IP/IPv6 address and optional device). Particularly, the entry does not store any link layer address. It always advertises the station address of the interface on which it sends advertisements (i.e. it's own station address).

C Route NAT status

NAT (or "Network Address Translation") remaps some parts of the IP address space into other ones. Linux-2.2 route NAT is supposed to be used to facilitate policy routing by rewriting addresses to other routing domains or to help while renumbering sites to another prefix.

What it is not: It is necessary to emphasize that *it is not supposed* to be used to compress address space or to split load. This is not missing functionality but a design principle. Route NAT is *stateless*. It does not hold any state about translated sessions. This means that it handles any number of sessions flawlessly. But it also means that it is *static*. It cannot detect the moment when the last TCP client stops using an address. For the same reason, it will not help to split load between several servers.

> NB. It is a pretty commonly held belief that it is useful to split load between several servers with NAT. This is a mistake. All you get from this is the requirement that the router keep the state of all the TCP connections going via it. Well, if the router is so powerful, run apache on it. 8)

The second feature: it does not touch packet payload, does not try to "improve" broken protocols by looking through its data and mangling it. It mangles IP addresses, only IP addresses and nothing but IP addresses. This also, is not missing any functionality.

To resume: if you need to compress address space or keep active FTP clients happy, your choice is not route NAT but masquerading, port forwarding, NAPT etc.

NB. By the way, you may also want to look at http://www.suse.com/m̃ha/HyperNews/get/linux-ip-nat.html

How it works. Some part of the address space is reserved for dummy addresses which will look for all the world like some host addresses inside your network. No other hosts may use these addresses, however other routers may also be configured to translate them.

NB. A great advantage of route NAT is that it may be used not only in stub networks but in environments with arbitrarily complicated structure. It does not firewall, it *forwards*.

These addresses are selected by the `ip route` command (sec.7.1, p.23). F.e.

```
ip route add nat 192.203.80.144 via 193.233.7.83
```

states that the single address 192.203.80.144 is a dummy NAT address. For all the world it looks like a host address inside our network. For neighbouring hosts and routers it looks like the local address of the translating router. The router answers ARP for it, advertises this address as routed via it, *et al*. When the router receives a packet destined for 192.203.80.144, it replaces this address with 193.233.7.83 which is the address of some real host and forwards the packet. If you need to remap blocks of addresses, you may use a command like:

```
ip route add nat 192.203.80.192/26 via 193.233.7.64
```

This command will map a block of 63 addresses 192.203.80.192-255 to 193.233.7.64-127.

When an internal host (193.233.7.83 in the example above) sends something to the outer world and these packets are forwarded by our router, it should translate the source address 193.233.7.83 into 192.203.80.144. This task is solved by setting a special policy rule (sec.8.1, p.37):

```
ip rule add prio 320 from 193.233.7.83 nat 192.203.80.144
```

This rule says that the source address 193.233.7.83 should be translated into 192.203.80.144 before forwarding. It is important that the address after the `nat` keyword is some NAT address, declared by `ip route add nat`. If it is just a random address the router will not map to it.

NB. The exception is when the address is a local address of this router (or 0.0.0.0) and masquerading is configured in the linux-2.2 kernel. In this case the router will masquerade the packets as this address. If 0.0.0.0 is selected, the result is equivalent to one obtained with firewalling rules. Otherwise, you have the way to order Linux to masquerade to this fixed address. NAT mechanism used in linux-2.4 is more flexible than masquerading, so that this feature has lost meaning and disabled.

If the network has non-trivial internal structure, it is useful and even necessary to add rules disabling translation when a packet does not leave this network. Let us return to the example from sec.8.2 (p.39).

```
300: from 193.233.7.83 to 193.233.7.0/24 lookup main
310: from 193.233.7.83 to 192.203.80.0/24 lookup main
320: from 193.233.7.83 lookup inr.ruhep map-to 192.203.80.144
```

This block of rules causes normal forwarding when packets from 193.233.7.83 do not leave networks 193.233.7/24 and 192.203.80/24. Also, if the `inr.ruhep` table does not contain a route to the destination (which means that the routing domain owning addresses from 192.203.80/24 is dead), no translation will occur. Otherwise, the packets are translated.

How to only translate selected ports: If you only want to translate selected ports (f.e. http) and leave the rest intact, you may use `ipchains` to `fwmark` a class of packets. Suppose you did and all the packets from 193.233.7.83 destined for port 80 are marked with marker 0x1234 in input fwchain. In this case you may replace rule #320 with:

```
320: from 193.233.7.83 fwmark 1234 lookup main map-to 192.203.80.144
```

and translation will only be enabled for outgoing http requests.

D Example: minimal host setup

The following script gives an example of a fault safe setup of IP (and IPv6, if it is compiled into the kernel) in the common case of a node attached to a single broadcast network. A more advanced script, which may be used both on multihomed hosts and on routers, is described in the following section.

The utilities used in the script may be found in the directory ftp://ftp.inr.ac.ru/ip-routing/:

1. `ip` — package `iproute2`.
2. `arping` — package `iputils`.
3. `rdisc` — package `iputils`.

NB. It also refers to a DHCP client, `dhcpcd`. I should refrain from recommending a good DHCP client to use. All that I can say is that ISC `dhcp-2.0b1pl6` patched with the patch that can be found in the `dhcp.bootp.rarp` subdirectory of the same ftp site *does* work, at least on Ethernet and Token Ring.

```
#! /bin/bash

# Usage: ifone ADDRESS[/PREFIX-LENGTH] [DEVICE]
# Parameters:
# $1 — Static IP address, optionally followed by prefix length.
# $2 — Device name. If it is missing, eth0 is asssumed.
# F.e. ifone 193.233.7.90

dev=$2
: ${dev:=eth0}
ipaddr=

# Parse IP address, splitting prefix length.

if [ "$1" != "" ]; then
  ipaddr=${1%/*}
  if [ "$1" != "$ipaddr" ]; then
    pfxlen=${1#*/}
  fi
  : ${pfxlen:=24}
fi
pfx="${ipaddr}/${pfxlen}"

# Step 0 — enable loopback.
#
# This step is necessary on any networked box before attempt
# to configure any other device.

ip link set up dev lo
ip addr add 127.0.0.1/8 dev lo brd + scope host

# IPv6 autoconfigure themself on loopback.
#
# If user gave loopback as device, we add the address as alias and exit.

if [ "$dev" = "lo" ]; then
  if [ "$ipaddr" != "" -a  "$ipaddr" != "127.0.0.1" ]; then
    ip address add $ipaddr dev $dev
    exit $?
  fi
  exit 0
fi

# Step 1 — enable device $dev
```

```
if ! ip link set up dev $dev ; then
  echo "Cannot enable interface $dev. Aborting." 1>&2
  exit 1
fi
```

The interface is UP. IPv6 started stateless autoconfiguration itself,
and its configuration finishes here. However,
IP still needs some static preconfigured address.

```
if [ "$ipaddr" = "" ]; then
  echo "No address for $dev is configured, trying DHCP..." 1>&2
  dhcpcd
  exit $?
fi
```

Step 2 — IP Duplicate Address Detection [9].
Send two probes and wait for result for 3 seconds.
If the interface opens slower f.e. due to long media detection,
you want to increase the timeout.

```
if ! arping -q -c 2 -w 3 -D -I $dev $ipaddr ; then
  echo "Address $ipaddr is busy, trying DHCP..." 1>&2
  dhcpcd
  exit $?
fi
```

OK, the address is unique, we may add it on the interface.
#
Step 3 — Configure the address on the interface.

```
if ! ip address add $pfx brd + dev $dev; then
  echo "Failed to add $pfx on $dev, trying DHCP..." 1>&2
  dhcpcd
  exit $?
fi
```

Step 4 — Announce our presence on the link.

```
arping -A -c 1 -I $dev $ipaddr
noarp=$?
( sleep 2;
  arping -U -c 1 -I $dev $ipaddr ) >& /dev/null </dev/null &
```

Step 5 (optional) — Add some control routes.
#
1. Prohibit link local multicast addresses.
2. Prohibit link local (alias, limited) broadcast.
3. Add default multicast route.

```
ip route add unreachable 224.0.0.0/24
ip route add unreachable 255.255.255.255
if [ ‘ip link ls $dev | grep -c MULTICAST‘ -ge 1 ]; then
  ip route add 224.0.0.0/4 dev $dev scope global
fi
```

Step 6 — Add fallback default route with huge metric.
If a proxy ARP server is present on the interface, we will be
able to talk to all the Internet without further configuration.
It is not so cheap though and we still hope that this route
will be overridden by more correct one by rdisc.
Do not make this step if the device is not ARPable,
because dead nexthop detection does not work on them.

```
if [ "$noarp" = "0" ]; then
  ip ro add default dev $dev metric 30000 scope global
fi
```

Step 7 — Restart router discovery and exit.

```
killall -HUP rdisc || rdisc -fs
exit 0
```

E Example: `ifcfg` — interface address management

This is a simplistic script replacing one option of `ifconfig`, namely, IP address management. It not only adds addresses, but also carries out Duplicate Address Detection [9], sends unsolicited ARP to update the caches of other hosts sharing the interface, adds some control routes and restarts Router Discovery when it is necessary.

I strongly recommend using it *instead* of `ifconfig` both on hosts and on routers.

```
#! /bin/bash
```

\# **Usage:** `ifcfg DEVICE[:ALIAS] [add|del] ADDRESS[/LENGTH] [PEER]`
\# **Parameters:**
\# —Device name. It may have alias suffix, separated by colon.
\# —Command: add, delete or stop.
\# —IP address, optionally followed by prefix length.
\# —Optional peer address for pointopoint interfaces.
\# F.e. `ifcfg eth0 193.233.7.90/24`
\# This function determines, whether it is router or host.
\# It returns 0, if the host is apparently not router.

```
CheckForwarding () {
  local sbase fwd
  sbase=/proc/sys/net/ipv4/conf
  fwd=0
  if [ -d $sbase ]; then
    for dir in $sbase/*/forwarding; do
      fwd=$[$fwd + `cat $dir`]
    done
  else
    fwd=2
  fi
  return $fwd
}
```

\# This function restarts Router Discovery.

```
RestartRDISC () {
  killall -HUP rdisc || rdisc -fs
}
```

\# Calculate ABC "natural" mask length
\# Arg: $1 = dotquad address

```
ABCMaskLen () {
  local class;
  class=${1%%.*}
  if [ $class -eq 0 -o $class -ge 224 ]; then return 0
  elif [ $class -ge 192 ]; then return 24
  elif [ $class -ge 128 ]; then return 16
  else  return 8 ; fi
}
```

\# **MAIN()**
\#
\# Strip alias suffix separated by colon.

```
label="label $1"
ldev=$1
dev=${1%:*}
if [ "$dev" = "" -o "$1" = "help" ]; then
  echo "Usage: ifcfg DEV [[add|del [ADDR[/LEN]] [PEER] | stop]" 1>&2
  echo "       add - add new address" 1>&2
  echo "       del - delete address" 1>&2
  echo "       stop - completely disable IP" 1>&2
  exit 1
fi
shift

CheckForwarding
fwd=$?
```

Parse command. If it is "stop", flush and exit.

```
deleting=0
case "$1" in
add) shift ;;
stop)
  if [ "$ldev" != "$dev" ]; then
    echo "Cannot stop alias $ldev" 1>&2
    exit 1;
  fi
  ip -4 addr flush dev $dev $label || exit 1
  if [ $fwd -eq 0 ]; then RestartRDISC; fi
  exit 0 ;;
del*)
  deleting=1; shift ;;
*)
esac
```

Parse prefix, split prefix length, separated by slash.

```
ipaddr=
pfxlen=
if [ "$1" != "" ]; then
  ipaddr=${1%/*}
  if [ "$1" != "$ipaddr" ]; then
    pfxlen=${1#*/}
  fi
  if [ "$ipaddr" = "" ]; then
```

```
    echo "$1 is bad IP address." 1>&2
    exit 1
  fi
fi
shift
```

\# If peer address is present, prefix length is 32.
\# Otherwise, if prefix length was not given, guess it.

```
peer=$1
if [ "$peer" != "" ]; then
  if [ "$pfxlen" != "" -a "$pfxlen" != "32" ]; then
    echo "Peer address with non-trivial netmask." 1>&2
    exit 1
  fi
  pfx="$ipaddr peer $peer"
else
  if [ "$pfxlen" = "" ]; then
    ABCMaskLen $ipaddr
    pfxlen=$?
  fi
  pfx="$ipaddr/$pfxlen"
fi
if [ "$ldev" = "$dev" -a "$ipaddr" != "" ]; then
  label=
fi
```

\# If deletion was requested, delete the address and restart RDISC

```
if [ $deleting -ne 0 ]; then
  ip addr del $pfx dev $dev $label || exit 1
  if [ $fwd -eq 0 ]; then RestartRDISC; fi
  exit 0
fi
```

\# Start interface initialization.
\#
\# **Step 0** — enable device `$dev`

```
if ! ip link set up dev $dev ; then
  echo "Error: cannot enable interface $dev." 1>&2
  exit 1
fi
if [ "$ipaddr" = "" ]; then exit 0; fi
```

```
# Step 1 — IP Duplicate Address Detection [9].
# Send two probes and wait for result for 3 seconds.
# If the interface opens slower f.e. due to long media detection,
# you want to increase the timeout.

if ! arping -q -c 2 -w 3 -D -I $dev $ipaddr ; then
  echo "Error: some host already uses address $ipaddr on $dev." 1>&2
  exit 1
fi

# OK, the address is unique. We may add it to the interface.
#
# Step 2 — Configure the address on the interface.

if ! ip address add $pfx brd + dev $dev $label; then
  echo "Error: failed to add $pfx on $dev." 1>&2
  exit 1
fi

# Step 3 — Announce our presence on the link

arping -q -A -c 1 -I $dev $ipaddr
noarp=$?
( sleep 2 ;
  arping -q -U -c 1 -I $dev $ipaddr ) >& /dev/null </dev/null &

# Step 4 (optional) — Add some control routes.
#
# 1. Prohibit link local multicast addresses.
# 2. Prohibit link local (alias, limited) broadcast.
# 3. Add default multicast route.

ip route add unreachable 224.0.0.0/24 >& /dev/null
ip route add unreachable 255.255.255.255 >& /dev/null
if [ `ip link ls $dev | grep -c MULTICAST` -ge 1 ]; then
  ip route add 224.0.0.0/4 dev $dev scope global >& /dev/null
fi

# Step 5 — Add fallback default route with huge metric.
# If a proxy ARP server is present on the interface, we will be
# able to talk to all the Internet without further configuration.
# Do not make this step on router or if the device is not ARPable.
# because dead nexthop detection does not work on them.
```

```
if [ $fwd -eq 0 ]; then
  if [ $noarp -eq 0 ]; then
    ip ro append default dev $dev metric 30000 scope global
  elif [ "$peer" != "" ]; then
    if ping -q -c 2 -w 4 $peer ; then
      ip ro append default via $peer dev $dev metric 30001
    fi
  fi
  RestartRDISC
fi

exit 0
```

End of **MAIN()**