IP Command Reference.

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

Contents

1 About this document. **3**

2 ip **— command syntax.** **3**

3 ip **— error messages.** **5**

4 ip link **— network device configuration.** **6**
4.1 ip link set — change device attributes. 6
4.2 ip link show — look at device attributes. 7

5 ip address **— protocol address management.** **11**
5.1 ip address add — add new protocol address. 12
5.2 ip address delete — delete protocol address. 13
5.3 ip address show — look at protocol addresses. 13
5.4 ip address flush — flush protocol addresses. 15

6 ip neighbour **— neighbour/arp tables management.** **16**
6.1 ip neighbour add — add new neighbour entry
ip neighbour change — change existing entry
ip neighbour replace — add new or change existing one. 17
6.2 ip neighbour delete — delete neighbour entry. 18
6.3 ip neighbour show — list neighbour entries. 18
6.4 ip neighbour flush — flush neighbour entries. 20

7 ip route **— routing table management.** **21**
7.1 ip route add — add new route
ip route change — change route
ip route replace — change route or add new one. 23
7.2 ip route delete — delete route. 27
7.3 ip route show — list routes. 27
7.4 ip route flush — flush routing tables. 31
7.5 ip route get — get single route. 32

8 `ip rule` — routing policy database management. **35**
8.1 `ip rule add` — insert new rule
`ip rule delete` — delete rule. 37
8.2 `ip rule show` — list rules. 39

9 `ip maddress` — multicast addresses management. **39**
9.1 `ip maddress show` — list multicast addresses. 40
9.2 `ip maddress add` — add multicast address
`ip maddress delete` — delete multicast address. 40

10 `ip mroute` — multicast routing cache management. **41**
10.1 `ip mroute show` — list mroute cache entries. 41

11 `ip tunnel` — tunnel configuration. **42**
11.1 `ip tunnel add` — add new tunnel
`ip tunnel change` — change existing tunnel
`ip tunnel delete` — destroy a tunnel. 43
11.2 `ip tunnel show` — list tunnels. 44

12 `ip monitor` and `rtmon` — state monitoring. **45**

13 Route realms and policy propagation, `rtacct`. **46**

References **47**

Appendix **48**

A Source address selection. **48**

B Proxy ARP/NDISC. **49**

C Route NAT status. **50**

D Example: minimal host setup. **51**

E Example: `ifcfg` — interface address management. **55**

1 About this document.

This document presents comprehensive description of `ip` utility from `iproute2` package. It is not tutorial and even not user guide, it is rather *dictionary* not explaining terms, but translating them to another ones, which also may be unknown to reader. However, the document is self-contained and provided the reader has basic networking background, he will find enough information and examples to understand and to configure Linux-2.2 IP and IPv6 networking.

The document is split to the sections explaining `ip` commands and options, decrypting the `ip` output and containing a few of examples. More voluminous examples and some topics, which require more elaborated discussion, are moved to appendix.

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

2 `ip` — command syntax.

Generic form of `ip` command is:

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

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

- `-V`, `-Version`

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

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

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

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

  — enforce protocol family to use. If the option is not present, protocol family is guessed from another arguments. If the rest of command line does not give enough information to guess family, `ip` falls back to default one, usually to `inet` or to `any`. `link` is 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 in single line, replacing line feeds with character `'\'`. It is convenient when you want to count records with `wc` or to `grep` the output. The trivial script `rtpr` converts the output back to readable form.

- `-r`, `-resolve`

  — allow to use system name resolver to print DNS names instead of host addresses.

  > NB. Do not use this option, when reporting bugs or querying for an advice.

  > NB. `ip` never tries to use DNS to resolve names to addresses.

`OBJECT` is object to manage or to get information about. The object types understood by current `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, names of all the objects may be written both in full and abbreviated form, f.e. `address` is abbreviated as `addr` or just `a`.

`COMMAND` specifies action to perform on the object. The set of possible actions depends on object type, as rule it is possible to `add`, to `delete` and to `show` (or to `list`) the object(s), but some objects allow not all these operations or have some additional commands. Command `help` is available for all the objects, it prints out 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 list of arguments to the command. The arguments depend on command and object. There are two types of arguments: *flags*, abbreviated with 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 default for `ip link` command, so that `ip link ls eth0` is equivalent to `ip link ls dev eth0`. In command descriptions following below such parameters are distinguished with marker "(default)".

Almost all the 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 an advice. "Officially" allowed abbreviations are listed in the document body.

3 `ip` — error messages.

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

- Wrong syntax of command line: an unknown keyword, wrongly formatted IP address *et al*. In this case `ip` exits not doing any actions, but printing some error message, as rule containing an information about the reason of the failure. Sometimes it prints also help page.
- The arguments did not pass verification for self-consistency.
- `ip` failed to compile kernel request from the arguments, because user gave not enough information.
- 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 syscall identifier.
- 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 part of device parameters ordered by command line.

It is difficult to list all the error messages (especially, about syntax errors), however as rule their meaning is clear in 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. Option `CONFIG_IP_MULTIPLE_TABLES` was not selected when configuring kernel. In this case any attempt to use command `ip rule` will fail, f.e.

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

4 `ip link` — network device configuration.

Object: `link` is a network device and the corresponding commands allow to look at the state of devices and to change it.

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

4.1 `ip link set` — change device attributes.

Abbreviations: `set`, `s`.

Arguments:

- `dev NAME` (default)

  — `NAME` specifies network device to operate on.

- `up` and `down`

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

- `arp on` or `arp off`

  — change `NOARP` flag on the device.

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

- `multicast on` or `multicast off`

  — change `MULTICAST` flag on the device.

- `dynamic on` or `dynamic off`

  — change `DYNAMIC` flag on the device.

- `name NAME`

  — change 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 transmit queue length of the device.

- `mtu NUMBER`

  — change MTU of the device.

- `address LLADDRESS`

  — change station address of the interface.

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

  — change link layer broadcast address or peer address in the case if the interface is `POINTOPOINT`.

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

NB. `ip` utility does not allow to change `PROMISC` and `ALLMULTI` flags, these flags are considered as obsolete and should not be changed administratively.

Warning: If multiple parameter changes are requested, `ip` aborts immediately after the change of at least one of them has failed. It is the only case, when `ip` can move the system to an unpredictable state. The solution is to avoid to change several parameters by one `ip link set` call.

Examples:

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

  — change station address of the interface `dummy`.

- `ip link set dummy up`

  — start the interface `dummy`.

4.2 `ip link show` — look at device attributes.

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

Arguments:

- `dev NAME` (default)

  — `NAME` specifies network device to show. If this argument is omitted, the command lists all the devices.

- `up`

  — display only 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 followed by colon is *interface index* or *ifindex*. This number uniquely identifies the interface. Then *interface name* follows (`eth0`, `sit0` etc.). Interface name is also unique at every given moment, however interface may disappear from the list (f.e. when corresponding driver module is unloaded) and another one with the same name will be recreated later. Besides that, administrator may change 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 sign `@`. It means that this device is bound to some another 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 interface *mtu* ("maximal transfer unit"), it determines maximal size of data, which can be sent as single packet over this interface.

qdisc ("queuing discipline") shows 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 the packets sent to it are immediately discarded. *qlen* is default transmit queue length of the device measured in packets.

In angle brackets interface flags are summarized.

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

- `LOOPBACK` — the interface does not communicate to another hosts, all the packets, which are sent through it, will be returned back and nothing but bounced back packets can be received.

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

- `POINTOPOINT` — the link has only two ends and two nodes attached to it. All the packets sent to the link will reach the peer and all the packets received by us are origined by this single peer.

  If neither `LOOPBACK` nor `BROADCAST` nor `POINTOPOINT` are set, the interface is supposed to be NMBA (Non-Broadcast Multi-Access). It is the most generic type of devices 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 advisory flag marking that the interface is aware of multicasting i.e. sending packets to some subset of neighbouring nodes. Broadcasting is particular case of multicasting, when multicast group consists of all the nodes on the link. It is important to emphasize that software *must not* interpret absence of this flag as impossibility to use multicasting on the 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, use of high bandwidth multicast transfers is not recommended on broadcast-only links because of high expenses, but it is not strictly prohibited.

- `PROMISC` — the device listens and feeds to kernel all the traffic on the link even if it is not destined to us, not broadcasted and not destined to a multicast group, which we are member of. Usually this mode exists only on broadcast links and 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 of another ones. It has no invariant value and its interpretation depends on network protocols involved. As rule it indicates that the device need not any address resolution and software or hardware know, how to deliver packets without any help from protocol stacks.

- `DYNAMIC` — is advisory flag marking this interface as dynamically created and destroyed.

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

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

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

The second line contains information on link layer addresses, associated with the device. The first word (`ether`, `sit`) defines interface hardware type. This type determines format and semantics of the addresses and logically it is part of the address. Default format of station address and broadcast address (or peer address for pointopoint links) is 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 address of broadcast relay or about address of ARP server.

NB. Multicast addresses are not shown by this command, see `ip maddr ls` in Sec.9 (p.39 of this document).

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

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

If the option `-s` 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, another devices may have these fields not zero, but they may have different interpretation.

5 `ip address` — protocol address management.

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

Object: `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 allows to look at addresses and their properties, to add new addresses and to delete old ones.

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

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

Abbreviations: `add`, `a`.

Arguments:

- `dev NAME`

  — name of the device to add the address.

- `local ADDRESS` (default)

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

- `peer ADDRESS`

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

- `broadcast ADDRESS`

  — broadcast address on the interface.

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

  > NB. Unlike `ifconfig` `ip` utility *does not* set any broadcast address, if it was not requested explicitly.

- `label NAME`

  — Each address may be tagged with 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 device name followed by colon.

- `scope SCOPE_VALUE`

  — 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 usual loopback address to loopback device.

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

  — add 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 protocol address.

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

Arguments: coincide with arguments of `ip addr add`. The device name is 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 loopback address from loopback device. It would be better not to try 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` — look at protocol addresses.

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

Arguments:

- `dev NAME` (default)

  — name of the device.

- `scope SCOPE_VAL`

  — list only addresses with this scope.

- `to PREFIX`

  — list only addresses matching this prefix.

- `label PATTERN`

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

- `dynamic` and `permanent`

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

- `tentative`

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

- `deprecated`

  — (IPv6 only) list only deprecated addresses.

- `primary` and `secondary`

  — list only 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 output of `ip link ls`, it is natural to interpret link layer addresses as addresses of protocol family `AF_PACKET`.

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

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

1. `secondary`

   — the address is not used, when selecting 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 leader of group of all the secondary addresses. When the leader is deleted, all the secondaries are purged too.

2. `dynamic`

   — the address was created due to stateless autoconfiguration [2]. In this case output contains also information on times, when the address is still valid. After `preferred_lft` expires the address is moved to deprecated state and 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 commands flushes protocol addresses selected 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.

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

Statistics: With the option `-statistics` the command becomes verbose, it prints out number of deleted addresses and number of rounds made to flush the address list. If the 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 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 of hosts sharing the same link. Neighbour entries are organized to tables, IPv4 neighbour table known under other name — ARP table.

The corresponding commands allow to look at neighbour bindings and their properties, to add new neighbour entries and to 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 `ip` utility.

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

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

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

Arguments:

- `to ADDRESS` (default)

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

- `dev NAME`

  — the interface which this neighbour is attached to.

- `lladdr LLADDRESS`

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

- `nud NUD_STATE`

  — state of the neighbour entry. `nud` is 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 removed only 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 reachability timeout expires.
  4. `stale` — the neighbour entry is valid, but suspicious. This option to `ip neigh` does not change 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 permanent ARP entry for 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 neighbour entry.

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

Description: This command invalidates a neighbour entry.

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

Example:

- `ip neigh del 10.0.0.3 dev eth0`

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

  NB. Deleted neighbour entry will not disappear from the tables immediately; if it is in use it cannot be deleted until the last client will release it, otherwise it will be destroyed during the next garbage collection.

Warning: attempts to delete or to change manually a `noarp` entry created by kernel may result in unpredictable behaviour. Particularly, kernel may start to try to resolve this address even on `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)

  — prefix selecting neighbours to list.

- `dev NAME`

  — list only neighbours attached to this device.

- `unused`

  — list only neighbours, which are not in use now.

- `nud NUD_STATE`

  — list only neighbour entries in this state. `NUD_STATE` takes values listed below or special value `all`, which means all the states. This option may occur more than once. If this option is absent, `ip` lists all the 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 protocol address of the neighbour, then device name follows. The rest of the line describes contents of neighbour entry identified by the pair (device, address).

`lladdr` is link layer address of the neighbour.

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

1. `none` — state of the neighbour is void.
2. `incomplete` — the neighbour is in process of resolution.
3. `reachable` — the neighbour is valid and apparently reachable.
4. `stale` — the neighbour is valid, but probably it is already unreachable, so that kernel will try to check it at the first transmission.
5. `delay` — a packet has been sent to the stale neighbour, kernel waits for confirmation.
6. `probe` — delay timer expired, but no confirmation was received. Kernel has started to probe 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 `noarp` entry, but only administrator may remove the entry from neighbour table.

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

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

Statistics: Option `-statistics` allows to look at 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 number of users of this entry, and `used` is 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 commands 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 default neighbour states to be flushed do not include `permanent` and `noarp`.

Statistics: With the option `-statistics` the command becomes verbose, it prints out number of deleted neighbours and number of rounds made to flush 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 kernel routing tables, which keep the information about paths to another networked nodes.

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

If several routes match to 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 TOS of some route with the longest prefix is equal to 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, route lookup fails.

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

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

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

   For simplicity we will limit ourselves to the case, when such situation is impossible and routes are uniquely identified by triplet {prefix, tos, preference}. Actually, it is impossible to create not-unique routes with `ip` commands described in this section.

   One useful exception to this rule is 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 advices from transport protocols to select working router, so that order of the routes is not essential. However, in this case it is not recommended to fiddle with default routes manually, but to use Router Discovery protocol (see Appendix D, p.51) instead. Actually, Linux-2.2 IPv6 even does not give user level applications any access to default routes.

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

Route attributes: Each route key refers to a routing information record, containing data required to deliver IP packets (f.e. output device and next hop router) and some optional attributes (f.e. path MTU or source address, preferred when communicating to 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 route *type*. The most important route type is `unicast` route; it describes real paths to another hosts. As rule, common routing tables contain only such routes. However, there exist another 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 route prefix.
- `unreachable` — these destinations are unreachable; packets are discarded and ICMP message *host unreachable* is generated. The local senders get error `EHOSTUNREACH`.
- `blackhole` — these destinations are unreachable; packets are discarded silently. The local senders get error `EINVAL`.
- `prohibit` — these destinations are unreachable; packets are discarded and ICMP message *communication administratively prohibited* is generated. The local senders get error `EACCES`.
- `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` — special control route used together with policy rules (see sec.8, p.35). If such route is selected, lookup in this table is terminated pretending that no route was found. Without policy routing it is equivalent to absence of the route in routing table, the packets are dropped and ICMP message *net unreachable* is generated. The local senders get error `ENETUNREACH`.

- `nat` — special NAT route. Destinations covered by the prefix are considered as 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 to be used as source address of any packet.

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

Route tables: Linux-2.2 can pack routes to several routing tables identified by 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 to the table `main` (ID 254) and kernel uses only this table, when calculating routes.

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

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

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

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

Arguments:

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

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

- `tos TOS` or `dsfield TOS`

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

- `metric NUMBER` or `preference NUMBER`

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

- `table TABLEID`

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

- `dev NAME`

  — the output device name.

- `via ADDRESS`

  — the address of nexthop router. Actually, the sense of this field depends on route type. For normal `unicast` routes it is either true nexthop router or, if it is 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 block of translated IP destinations.

- `src ADDRESS`

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

- `realm REALMID`

  — the realm which this route is assigned to. `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 destination. If modifier `lock` is not used, 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 the packets will be sent without 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, which our TCP peers are allowed to send to us.

- `rtt NUMBER`

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

- `rttvar NUMBER`

  — [2.3.15+ only] initial RTT variance estimate.

- `ssthresh NUMBER`

  — [2.3.15+ only] estimate for inititial slow start threshould.

- `cwnd NUMBER`

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

- `advmss NUMBER`

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

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

- `nexthop NEXTHOP`

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

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

- `scope SCOPE_VAL`

  — 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 routes and broadcasts and scope `host` for `local` routes.

- `protocol RTPROTO`

  — 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 considers the route is added by someone, who does not understand what he does). Several protocol values have a fixed interpretation. Namely:

  - `redirect` — route was installed due to ICMP redirect.

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

The rest of values are not reserved and 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 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 route will be frozen to one selected nexthop, so that load splitting will occur only on per-flow base. `equalize` works only if the kernel is patched.

NB. Actually there exist more commands: `prepend` does the same thing as classic `route add`, i.e. adds 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 to use these features.

NB. More sad news, IPv6 understands only `append` command correctly, all the rest of the set translating to `append`. Certainly, it will change in the future.

Examples:

- add 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 direct route via device `dummy`

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

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

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

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

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

```
ip route add nat 192.203.80.142 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 route.

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

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

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

NB. Linux-2.0 had option to delete route selected only by prefix address ignoring its length (i.e. netmask). This option does not exist more, because it was ambiguous. However, look at `ip route flush` (sec.7.4, p.31), it provides similar and even more rich functionality.

Example:

- delete multipath route created by 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 allows to view routing tables contents or to look at the route(s) selected by some criteria.

Arguments:

- `to SELECTOR` (default)

  — select routes only from given range of destinations. `SELECTOR` consists of 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 all the 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 exactly with this prefix. If neither of these options are present, `ip` assumes `root 0/0` i.e. it lists all the table.

- `tos TOS` or `dsfield TOS`

  — Select only routes with given TOS.

- `table TABLEID`

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

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

    NB. IPv6 has single table, however splitting to `main`, `local` and `cache` is emulated by `ip` utility.

- `cloned` or `cached`

  — list cloned routes i.e. routes, which were dynamically forked of another 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 bounds source address range rather than destinations. Note, that `from` option works only with cloned routes.

- `protocol RTPROTO`

  — list only routes of this protocol.

- `scope SCOPE_VAL`

  — list only routes with this scope.

- `type TYPE`

  — list only routes of this type.

- `dev NAME`

  — list only routes going via this device.

- `via PREFIX`

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

- `src PREFIX`

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

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

  — list only 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 size of routing cache we have to use option `-o`, because cached attributes can take more than one line of the 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, it is the case when the route is cloned or you requested additional statistics. If the option `-o` was given, then line feeds separating lines inside records are replaced with 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 another attributes, which are evaluated during route calculation and updated during route lifetime. The 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, does not it? Did you notice, that it is 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, started with the word `cache`, shows additional attributes, which normal routes do not possess. In angle brackets cached flags are summarized:

- `local` — packets are delivered locally. It stands for loopback unicast routes, for broadcast routes and for multicast routes, if this host is member of the corresponding group.
- `reject` — the path is bad. Any attempt to use it results in 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 ICMP redirect.
- `fastroute` — route is eligible to be used for fastroute.
- `equalize` — make packet by packet randomization along this path.
- `dst-nat` — destination address requires translation.
- `src-nat` — source address requires translation.
- `masq` — source address requires masquerading.
- `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 to 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 option `-statistics` more information about this route is shown:

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

7.4 `ip route flush` — flush routing tables.

Abbreviations: `flush`, `f`.

Description: the command allows to flush 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 different thing is default action: if `show` dumps all the IP main routing table, `flush` prints helper page. The reason of this difference does not require any explanation, does it?

Statistics: With the option `-statistics` the commands becomes verbose, it prints out number of deleted routes and number of rounds made to flush 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 main table (f.e. after routing daemon crash).

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

This option deserved to be put into scriptlet `routef`.

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

The second example is flushing 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 is flushing BGP routing tables after `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 single route.

Abbreviations: `get`, `g`.

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

Arguments:

- `to ADDRESS` (default)

  — destination address.

- `from ADDRESS`

  — source address.

- `tos TOS` or `dsfield TOS`

  — Type Of Service.

- `iif NAME`

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

- `oif NAME`

  — enforce output device, which this packet will be routed out.

- `connected`

  — if no source address (option `from`) was given, relookup route with source set to preferred address, received from the first lookup. If policy routing is used, it may be 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 it is necessary. Essentially, `get` is equivalent to sending a packet along this path. If argument `iif` is not given the kernel creates route to output packets towards requested destination. This is equivalent to pinging this destination with subsequent `ip route ls cache`, however no packets are sent really. With the argument `iif` kernel pretends that a packet arrived from this interface and searches for path to forward the packet.

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

Examples:

- Find 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 route to forward packets arriving on `eth0` from 193.233.7.82 and destined to 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. It is this operation that created funny route to 193.233.7.82 looped back to 193.233.7.82 (cf. NB on p.30). Note `redirect` flag on it.

- Find multicast route for packets arriving on `eth0` from host 193.233.7.82 and destined to multicast group 224.2.127.254 (it is supposed, 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 of ones seen before. It contains “normal” part and “multicast” part. Normal part is used to deliver (or not to deliver the packet) to local IP listeners. In this case the router is not member of this group, so that route has no `local` flag and only forwards packets. The output device for such entries is always loopback. Multicast part consists of additional `Oifs:` list showing output interfaces.

It is time for more complicated example. Let us add invalid gatewayed route for a destination, which really is 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 did occur? Router 193.233.7.254 understood that we have much better path to the destination and sent to us ICMP redirect message. We may retry `ip route get` to look, what we have in 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: Rules in routing policy database controlling 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 TOS field). 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 the destination addresses, but also on another 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 conventional destination based routing table, ordered according to the longest match rule, is replaced with “routing policy database” (or RPDB), which selects route executing some set of rules. The rules may have lots of keys of different nature and therefore they have no natural ordering, but imposed by administrator. Linux-2.2 RPDB is linear list of rules ordered by numeric priority value. RPDB explicitly allows to match a few of packet fields:

- packet source address.
- packet destination address.

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

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

Each routing policy 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 to the packet, the action is performed. The action predicate may return with success, in this case it will give either route or failure indication and RPDB lookup is terminated. Otherwise, RPDB program continues on the next rule.

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

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

1. Priority: 0, Selector: match anything, Action: lookup routing table `local` (ID 255). The table `local` is 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 table `main` is normal routing table, containing all not-policy routes. This rule may be deleted and/or overridden with another ones by administrator.

3. Priority: 32767, Selector: match anything, Action: lookup routing table `default` (ID 253). The table `default` is empty, it is reserved for some post-processing, if previous default rules did not select the packet. This rule also may be deleted.

Do not mix routing tables and 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 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 attached. F.e. each rule has pointer to some routing table. NAT and masquerading rules have attribute to select new IP address to translate/masquearade. Besides that, rules have some of optional attributes, which routes have, namely `realms`. These values do not override those contained in routing tables, they are used only if the route did not select any attributes.

Rule types: 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 drop packet silently.
- `unreachable` — the rule prescribes to generate error "Network is unreachable".
- `prohibit` — the rule prescribes to generate error "Communication is administratively prohibited".
- `nat` — the rule prescribes to translate source address of the IP packet to some another value. More about NAT is in Appendix C, p.50.

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

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

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

Arguments:

- `type TYPE` (default)

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

  — select source prefix to match.
- `to PREFIX`

  — select destination prefix to match.
- `iif NAME`

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

- `tos TOS` or `dsfield TOS`

  — select TOS value to match.

- `fwmark MARK`

  — select value of `fwmark` to match.

- `priority PREFERENCE`

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

  > NB. Really, by historical reasons `ip rule add` does not require any priority value and allows it to be not unique. If user did not supplied any priority, it is selected by kernel. If user requested to create rule with a priority value, which already exists, kernel does not reject the request and adds new rule before all old rules of the same priority.
  >
  > It is mistake in design, not more. And it will be fixed one day, so that do not rely on this feature, use explicit priorities.

- `table TABLEID`

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

- `realms FROM/TO`

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

- `nat ADDRESS`

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

Warning: Changes to RPDB made with these commands do not become active immediately. It is supposed that after script finishes batch of updates it flushes 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 193.233.7.83 to 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 unused default rule:

```
ip ru del prio 32767
```

8.2 `ip rule show` — list rules.

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

Arguments: Good news, it is the only command, which 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 position rule priority value stands followed by colon. Then the selectors follow, each key is prefixed by the same keyword, which was used to create the rule.

The keyword `lookup` is followed by 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 keyword `map-to` followed by start of 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 internal network, but they are routed differently, when the packets leave it. Besides that, the host 193.233.7.83 is translated to another prefix to look as 192.203.80.144, when talking to outer world.

9 `ip maddress` — multicast addresses management.

Object: The `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 interface index and its name. Then multicast address list follows, each line starts by protocol identifier. The word `link` denotes link layer multicast addresses.

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

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

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

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

Description: these commands allow to 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 manages only link layer addresses.

Arguments:

- `address LLADDRESS` (default)
  — link layer multicast address.

- `dev NAME`
  — the device to join/leave this multicast address.

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

```
netadm@alisa:~ # ip maddr add 33:33:00:00:00:01 dev dummy
netadm@alisa:~ # ip -0 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 kernel check for multicast address validity. Particularly, it means that you can try to load unicast address instead of multicast. The most of drivers will ignore such addresses, but several ones (f.e. Tulip) really intern it to their on-board filter. The effect may be strange. Namely, the addresses become additional local link addresses and, if you loaded address of another host to router, wait for duplicated packets on the wire. It is not a bug, it is rather hole in API and intra-kernel interfaces. This feature really more useful for traffic monitoring, but using it with Linux-2.2 you *have to* be sure, that the host is not router and, especially, it is not transparent proxy or masquerading agent.

10 `ip mroute` — multicast routing cache management.

Abbreviations: `mroute`, `mr`.

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

Due to limitations of current interface to multicast routing engine it is impossible to change `mroute` objects administratively, so that we may only look at 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)

  — prefix selecting destination multicast addresses to list.

- `iif NAME`

  — interface which multicast packets are received on.

- `from PREFIX`

  — prefix selecting IP source addresses of 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 multicast routing cache, where S is source address and G is multicast group. `Iif` is interface which multicast packets are expected to arrive on. If word `unresolved` stands instead of interface name, it means that routing daemon still did not resolve this entry. Keyword `oifs` is followed by list of output interfaces, separated by spaces. If multicast routing entry is created with non-trivial TTL scope, administrative distances are appended the device names in `oifs` list.

Statistics: Option `-statistics` also prints number of packets and bytes forwarded along this route and number of packets arrived on 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: The `tunnel` objects are tunnels, encapsulating packets in IPv4 packets and sending them over IP infrastructure then.

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

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

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

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

Arguments:

- `name NAME` (default)

  — select tunnel device name.

- `mode MODE`

  — set tunnel mode. Three modes are available now `ipip`, `sit` and `gre`.

- `remote ADDRESS`

  — set remote endpoint of the tunnel.

- `local ADDRESS`

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

- `ttl N`

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

- `tos T` or `dsfield T`

  — set fixed TOS `T` on tunneled packets. Default value is: `inherit`.

- `dev NAME`

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

- `nopmtudisc`

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

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

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

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

  — (only GRE tunnels) checksum tunneled packets. The flag `ocsum` orders to checksum outgoing packets, `icsum` requires that all the input packets had correct checksum. `csum` is equivalent to the combination "`icsum ocsum`".

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

  — (only GRE tunnels) serialize packets. The flag `oseq` enables sequencing outgouing packets, `iseq` requires that all the input packets were serialized. `seq` 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 even do not understand, how it is supposed to work and for what purpose Cisco planned to use it. Do not use it.

Example: create 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:

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 tunnel device name terminated by colon, then tunnel mode follows. The parameters of the tunnel are listed with the same keywords which were used at tunnel creation.

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 numbers, which are printed with `ip -s link show` (sec.4.2, p.7), but tags are different to reflect tunnel specific.

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

12 ip monitor and rtmon — state monitoring.

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

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

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

If a file name is given, it does not listen RTNETLINK, but opens the file containing RTNETLINK messages saved in binary format and dumps them. Such history file can be generated with utility `rtmon`. This utility has command line syntax similar to one of `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 start.

13 Route realms and policy propagation, `rtacct`.

On routers using OSPF ASE or, especially, BGP protocol routing tables may be huge. If we want to classify or to account 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 impossible physically.

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

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

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

Namely, routes can be clustered together at user space, based on their attributes. F.e. BGP router knows route ASPATH, its community; OSPF router knows route tag or its area. Administrator, when adding routes manually, also knows their nature. Provided 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 supposed, that this identification is made by routing daemon, but static routes also can be handled manually with `ip route` (see sec.7, p.21).

> NB. There exists patch to `gated`, allowing to classify 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 the case, when routing daemon is not aware of realms), missing realms may be completed with routing policy rules, see sec. 8, p.35.

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

1. If route has a realm, destination realm of the packet is set to it.
2. If rule has a source realm, source realm of the packet is set to it. If destination realm was not get from route and rule has destination realm, it is also set.
3. If at least one of realms is still unknown, kernel finds reversed route to the source of the packet.
4. If source realm is still unknown, get it from reversed route.
5. If one of realms is still unknown, swap 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 realms is unknown, it is initialized to zero (or realm `unknown`).

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

Much simpler, but still very useful application is ingres accounting by realms. Kernel gathers packet statistics summary, which can be viewed with utility `rtacct`.

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

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

Note that locally originated packets are not accounted here, `rtacct` shows ingres packets only. Using `route` classifier (see [7]) you can get even more detailed accounting information about outgres 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 originates an IP packet, it must select some source address. Correct source address selection is critical procedure, because it gives to the receiver the information, how to deliver reply. If the source is selected incorrectly, in the best case the backward path may appear different of forward one, which is harmful for performance. In the worst case, when the addresses are administratively scoped, the reply may be lost at all.

Linux-2.2 selects source addresses using the following algorithm:

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

  NB. Never say "Never". Sysctl option `ip_dynaddr` breaks this axiom. It has been made deliberately with purpose to reselect automatically 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 explicit source address hint for this destination. The hint is set with parameter `src` to `ip route` command, sec.7, p.21.

- Otherwise, the kernel searches through list of addresses, attached to the interface, which packets will be routed out. The search strategies are different for IP and IPv6. Namely:

  - IPv6 searches for the first valid, not deprecated address with the same scope as destination.
  - IP searches for the first valid address with scope wider than scope of the destination, but it prefers the 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 and they are supplied in routing tables instead (parameter `scope` to `ip route` command, sec.7, p.21).

- Otherwise, if the scope of destination is `link` or `host`, the algorithm fails and returns zero source address.

- Otherwise, all the interfaces are scanned to search for an address with appropriate scope. Loopback device `lo` is always the first in 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 another hosts. In linux-2.2 proxy ARP on an interface may be enabled by setting kernel `sysctl` variable `net/ipv4/conf/<dev>/proxy_arp` to 1. After this router starts to answer ARP requests on the interface `<dev>`, provided the route to the requested destination goes *not* back via the same device. The variable `net/ipv4/conf/all/proxy_arp` enables proxy ARP on all the IP devices.

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

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

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

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

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

adds new proxy ARP/NDISC record and

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

deletes it.

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

It is important to emphasize that proxy entries have *no* parameters different of these ones (IP/IPv6 address and optional device). Particularly, the entry does not store any link layer address, it always advertises its own station address of the interface, where it sends advertisements.

C Route NAT status.

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

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

> NB. It is pretty common belief that it is useful to split load between several servers with NAT. It is mistake. All that you get from this is that router have to keep 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. It is also not missing functionality.

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

> NB. By the way, you may look also at http://www.csn.tu-chemnitz.de/HyperNews/get/linux-ip-nat.html

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

> NB. It is great advantage of route NAT, 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 `ip route` command (sec.7.1, p.23). F.e.

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

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

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

This command will map block of 63 addresses 192...255 to 64...127.

When internal host (193.233.7.83 in the example above) send something to the outer world and these packets are forwarded by our router, it should translate source address 193.233.7.83 to 192.203.80.144. This task is solved by setting 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 source address 193.233.7.83 should be translated to 192.203.80.144 before forwarding. It is important that address standing after keyword `nat` was 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 masquearading is configured in the kernel. In this case 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.

If the network has non-trivial internal structure, it is useful and even necessary to add rules disbabling translation, when 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 orders to make normal forwarding, when packets from 193.233.7.83 do not leave networks 193.233.7/24 and 192.203.80/24. If the table `inr.ruhep` does not contain route to the destination (which means that routing domain owning addresses from 192.203.80/24 is dead), no translation also will occur. Otherwise, the packets are translated.

How to translate only selected ports. If you want to translate only selected ports (f.e. http) and leave all the rest intact, you may use `ipchains` to mark a class of packets with a `fwmark`. Suppose, you did it and all the packets from 193.233.7.83 destined to 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 be enabled only for outgouing http requests.

D Example: minimal host setup.

The script following below gives an example of fault safe setup of IP (and IPv6, if it is compiled into the kernel) in the common case of node attached to a single broadcast network. 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 DHCP client `dhcpcd`. I should refrain from attempt to recommend any good DHCP client to use. All that I can say is that ISC `dhcp-2.0b1pl6` patched by patch, which can be found in subdirectory `dhcp.bootp.rarp` 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 presense 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.

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

I strongly recommend to use it *instead* of `ifconfig` both on hosts and on routes.

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

\# **Usage:** `ifcfg DEVICE[:ALIAS] [add|del] ADDRESS[/LENGTH] [PEER]`
\# **Parameters:**
\# —Device name. It may have alias suffix, separted 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.
Otheriwise, 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 presense 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()**