MASARYKOVA UNIVERZITA
FAKULTA INFORMATIKY

Documentation for iproute

BACHELOR THESIS

Marek Andreánsky

Brno, fall 2010

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Marek Andreánsky

Advisor: Marek Grác

Acknowledgement

I would like to thank my advisors Marcela Mašláňová and Marek Grác.

Abstract

The aim of this bachelor thesis is to provide a simple English documentation to the iproute2 package. The guide should contain simple command usage examples and explain the networking capabilities that are present in the Linux kernel and can be configured by iproute2.

Keywords

iproute2, Linux, networking, tunnel, QoS, shaper, HTB, routing, IPsec, XFRM

Contents

1 The basics of iproute2

1.1 What is iproute2

The *iproute2 package* is a collection of *Linux* networking utilities. The package is present in most Linux distributions in their basic installation and can be used to administer every aspect of the Linux network stack.

1.1.1 History

The iproute2 package is a successor of *net-tools*. Net-tools is a collection of network management tools that were used for manipulating the Linux network stack in the 2.4.x series Linux kernels. Net-tools grew obsolete a few years ago and a replacement was needed. A replacement that could support the new networking functionality present in the latest Linux kernel while improving the usability of the applications contained within. This was achieved by presenting a more usable and user friendly command structure.

Iproute2 was originally created by *Alexey Kuznetsov*, who was responsible for the *QoS*[1] *implementation* in the Linux kernel. It is currently maintained by *Stephen Hemminger*, who is one of the Linux kernel networking stack maintainers. Iproute2 is being actively worked on in comparison with net-tools. Proof of active iproute2 development are the latest additions of *6rd*[2] *tunneling support* and *IPsec*[3] *management and configuration*.

1.1.2 Comparison with net-tools

The tools present in the iproute2 package can achieve the same level of control over the Linux network stack as net-tools. Compared to net-tools, it can also also configure advanced *routing* and *shaping* capabilities present in the latest versions of the Linux kernel while having a simpler and more unified notation of the commands needed for

1. quality of service, a resource reservation control mechanism
2. explained in chapter IPv6 rapid deployment (6rd)
3. explained in chapter IPsec

operation. The iproute2 ip command can manage most standard networking features present in the Linux kernel – it can add new entries to the *routing tables*[1], change the *ARP*[2] *or NDISC*[4] *cache* settings, add *network tunnels*, enable or disable a network interface, view *link layer information* or *manage* the *IP addresses* assigned to an interface, both for *IPv4* and *IPv6*.

The iproute2 package makes it easier for a newcomer to Linux networking, as it contains a unified set of tools that are aimed at easy network management. Net-tools also has not been actively worked on for several years and are also noted as obsolete by the Linux foundation[14], therefore there isn't any valid reason for a newcomer to use net-tools over iproute2.

1.1.3 Iproute2 installation

Iproute2 should be present and installed in most recent Linux distributions *by default*, as it is the bare component needed for Linux to do any kind of networking management. Iproute2 is required for the *network-manager Gnome*[5] *package*, meaning that removing it would usually render the machine unable to connect to the network when using the *Gnome desktop*. Resorting to net-tools would also allow the editing of networking, but it would be unable to use any latest networking kernel components which are one of the main benefits of using Linux as a *router*, *gateway* or any other kind of advanced network component.

1.1.4 Contents of the iproute2 package on Fedora 13

The contents of the iproute2 package are mostly identical on different distributions. Fedora 13 will be used in this chapter as an example because of it's *2.6.34 Linux kernel* and *iproute2-ss100224* package

1. data structure in the form of a table-like object stored in a router or a networked computer that lists the routes to particular network destinations
2. Address Resolution Protocol, used for determining a network neighbors MAC address[3], specified in RFC 826[22]
3. unique identifier assigned to a network interface
4. Neighbor Discovery, an IPv6 mechanism that replaces ARP[24]
5. GNU Network Object Model Environment, a Linux graphical user interface

that are present in the standard Fedora 13 repositories. It is therefore easiest for a new user to try the latest version of the package without the need to compile either the kernel or iproute2 itself.

Iproute2 contains several smaller applications, *tools*, that each aim to achieve a specific networking task in the Linux kernel. The most prominent of the tools is *ip* which contains the *ip address*, *ip addrlabel*, *ip link*, *ip maddr*, *ip ntable*, *ip route*, *ip rule ip tunnel* and *ip xfrm* subcommands.

Ip address is aimed at managing ip addresses assigned to network interfaces, both IPv4 and IPv6. *Ip addrlabel* can manage labels[1], *ip link* can access and change *link layer information*, *ip maddr* can manage multicast addressess, *ip ntable* can modify ARP and NDISC cache settings, *ip route* can manage the Linux *routing tables*, *ip tunnel* can create, edit and remove network tunnels and *ip xfrm* can be used to manage IPsec[2] on Linux.

There is also the *tc* tool, which can *control network traffic* by either *shaping* or *network emulation* of larger networks.

Each tool has its own chapter in which more details and examples of usage will be given.

The list of files included in the iproute2 package on *Fedora 13* can be gained by using *yum-utils*. They are not installed by default and can be installed with the command

```
yum install yum-utils
```

Finding and listing all the files included in the iproute2 Fedora package can be done with the command:

```
repoquery --list iproute
```

The output will list all the iproute2 files, the most prominent of them being noted below with basic information about their function. The more prominent files will be described later in more detail with simple usage examples.

1. explained in chapter Command: ip addrlabel
2. explained in chapter IPsec

`/etc/iproute2/ematch_map`
This file contains the extended match (ematch) map. An extended match is a small classification tool not worth writing a full classifier for. Ematches can be interconnected to form a logic expression and get attached to classifiers to extend their functionality. [15]

`/etc/iproute2/rt_dsfield`
This file contains the list of packet *DS fields*. A classifier can select packets depending on the value of their DS field[1].

`/etc/iproute2/rt_protos`
This file contains the list of network protocols.

`/etc/iproute2/rt_realms`
This file contains the list of realms. The default realm is 0, is named cosmos and is the realm that is used when the ingress or outgress realm is unknown.

Realms can be used to simplify routing, as it is easier to manage a few realms than a few hundred routes.

`/etc/iproute2/rt_scopes`
This file contains the list of scopes. Three values are reserved – *host*, *link* and *global*. Each of these scopes can be assigned to an networking interface manually, but iproute2 will assign one automatically depending on the type of the address.

When a scope is assigned, the interface will have a limited range depending on the type of scope selected.

Host scope limits the ip address to the current host only. The loopback interface always has a host scope.

Link scope limits the ip address to the local network, routers do not forward packets with the link scope ip address but assign a new address to the packets.

Global scope means the address can be used anywhere and can be forwarded by routers.

Site scope (IPv6 only) means that routers will not forward any

1. Differentiated Services field, a QoS classifying mechanism, specified in *RFC 2474*[5]

packet with site-local source or destination address outside the site.

`/etc/iproute2/rt_tables`
This file contains the *Linux routing tables*. It can be manually modified in a text editor to add or remove routing tables.

`/etc/sysconfig/cbq/avpkt`
Specifies the size of an average packet for CBQ[1].

`/etc/sysconfig/cbq/cbq-0000.example`
An example CBQ configuration file.

`/sbin/cbq`
This is a CBQ example script.

`/sbin/ifcfg`
This is a simplistic script that replaces the IP address management option of ifconfig.

`/sbin/ip`
The main iproute2 command that is used to manage most of the Linux kernel networking capabilities.

`/sbin/rtmon`
Rtmon (routing monitor) is used to listen on netlink sockets[2] and can monitor routing table changes.

`/sbin/tc`
A tool used for advanced traffic control management on Linux, such as network shaping or emulating larger scale networks.

`/usr/sbin/arpd`
This is the *userspace ARP daemon* that collects gratuitous ARP information, saving it on the local disk and feeding it to kernel on demand

1. Class Based Queueing, explained in chapter CBQ
2. socket-like mechanism between the kernel and user space processes or only between user space processes

to avoid redundant broadcasting due to the limited size of the kernel *ARP cache*.

`/usr/sbin/lnstat`
Lnstat is a generalized and more feature-complete replacement for the old rtstat command.

`/usr/sbin/nstat`
`/usr/sbin/rtacct`
Two network statistics tools.

`/usr/sbin/ss`
The ss command is used to dump socket statistics. It can show information similar to the netstat command of the now deprecated net-tools package.

`/usr/share/doc/iproute-2.6.33/README`
This directory contains several readme files that contain advice for installation, compilation or specific configuration of iproute2.

`/usr/share/man/man8/`
This directory contains the manual pages for all of the iproute2 tools, commands and algorithms, each assigned by its name.

`/usr/share/tc`
This directory contains the *normal*, *pareto* and *paretonormal* distributions that can be assigned to a *qdisc*[1] when simulating network delay.

1.2 Basic networking terminology used with iproute2

1.2.1 Network interface

From the Linux kernel's point of view, a *network interface* is a software object that can process outgoing packets, and the actual transmission mechanism remains hidden inside the interface driver.

1. Queuing discipline, chapter Queuing discipline

The Linux kernel can have *physical* or *logical* networking interfaces. A *physical* networking interface is linked to an existing physical networking card, while a *logical* network interface only exists inside the Linux kernel and is therefore not linked to any specific networking hardware. Both logical and physical networking interfaces can be managed to the same extent with iproute2.

1.2.2 Queuing discipline

A *queuing discipline* (qdisc in short) is a *scheduler*. A scheduler manages incoming and outgoing traffic on a network interface by using the algorithm that is assigned to it.

If no scheduler is assigned then a network interface can't work with incoming and outgoing packets as it does not know how and in what order to process them, therefore each network interface must have at least one scheduler assigned. There are several qdiscs present in the Linux kernel, the default scheduler (default meaning that it is used on every new network interface) being *pfifo_fast*, which will be explained in more detail and compared to other qdiscs in chapter Pfifo_fast.

If a network interface with no scheduler is enabled, pfifo_fast will be automatically assigned to it.

1.2.3 Traffic shaping

Traffic shaping, also known as packet shaping, is the practice of regulating network data transfer to assure a certain level of performance and *quality of service (QoS)*.

The practice involves delaying the flow of packets that have been designated as less important or less desired than those of prioritized traffic streams. Regulating the flow of packets into a network is known as *bandwidth throttling*. Regulation of the flow of packets out of a network is known as *rate limiting*.

Traffic shaping can be used to optimize and existing network to maximize it's potential bandwidth without the need of an additional investment in the form of new networking hardware or an improved Internet line from the Internet service provider (ISP). It can also be used to provide a certain guaranteed QoS to all of the hosts present

in the network by limiting others.

If a single host is stressing the available line by using most of the bandwidth a limit can be imposed on that specific host. This limit can be achieved by shaping network traffic that is coming to or from that host. This will not limit the other hosts but will improve their QoS.

Iproute2 can configure traffic shaping that is present inside the Linux kernel with the *tc (traffic control)* command of the iproute2 package. Use of the tc command will be explained in chapter Traffic control with TC.

1.2.4 Network Tunnel

A network tunnel is a *point to point connection* between two hosts that enables the transport of selected data packets through a network that does not necessarily use the same networking protocol. An example use of a network tunnel is transporting IPv6 packets over an IPv4 network.

Transporting IPv6 packets over an IPv4 network is impossible unless every networking hardware that the packet has to pass also has IPv6 support. As this cannot be assured, a network tunnel can be created between two end points that need to communicate using IPv6 to guarantee that the connection between those two points is always available. This is called a *6to4 network tunnel*.

There are different types of network tunnels that can be enabled on Linux but a 6to4 tunnel is the most used type and tunneling functionality will be therefore explained using a 6to4 tunnel example model.

An 6to4 tunnel *encapsulates* an IPv6 packet that is going to be sent through it into an IPv4 packet by adding an IPv4 header while preserving the whole IPv6 packet inside the body of the new IPv4 packet. The *payload* (an IPv6 packet in this case) is then sent through the tunnel and flows through the network until it reaches the end of the tunnel, where it will be *de-encapsulated*.

By de-encapsulating the IPv4 packet, the original IPv6 packet is gained. This packet can then safely flow through the IPv6 compatible network and reach its destination without the endpoint ever knowing it went through an IPv4 network.

This process can be applied on different combinations of proto-

cols and can connect hosts through networks that use incompatible protocols.

1.2.5 Routing

Routing is the process of selecting paths in a network along which to send network traffic.

Routing information is stored in the routing tables. New routing tables can be added by manually modifying the *rt_tables* file in a text editor. Each table can contain routing rules which route network traffic originating or going through the Linux host.

Routing rules can be added with the *ip route* command. Practical examples of using the command will be given in chapter Command: ip route.

1.2.6 IPsec

IPsec[1] is a protocol suite aimed at securing the communication over the Internet Protocol[2] (IP) by authenticating and encrypting every transmitted packet.

IPsec achieves security by using either the *Authentication Header*[3] (AH), *Encapsulating Security Payload*[4] (ESP) protocol or both at the same time.

AH offers data origin authentication and integrity and can optionally support anti-replay features. ESP offers confidentiality in addition to everything that AH does.

ESP support is required for an IPsec implementation to work, while AH is only recommended. This is due to the fact that there are few cases when ESP cannot provide the necessary security features which are desired, and also that ESP is used most of the time when implementing IPsec on Linux.

In order for IPsec to function properly between two hosts, a *Se-*

1. specified in *RFC 4301*[25]
2. the most common communications protocol used for relaying packets across a network
3. specified in *RFC4302*[18]
4. specified in *RFC4303*[19]

curity Association[6] (SA) and *Security Policy*[5] (SP) has to be created on each of the hosts.

1.2.6.1 Security association

A security association (SA) is a *simplex uni-directional connection* that affords security services to data transferred between two network interfaces.

If the interfaces wish to communicate *bi-directionally*, *two security associations* have to be *created*, one for each direction. If both Authentication Header and Encapsulating Security Payload protection are needed to be applied to a data stream, two security associations have to be created, one per each protocol.

Security associations are stored in the *Security Association Database* (SAD) that is present in the 2.6.34 kernel and can be managed using the *ip xfrm* tool. Examples of use are shown in chapter Command: ip xfrm.

A security association contains the *type of protocol* (ESP or AH) and *algorithm* (DES or 3DES for encryption; MD5 or SHA-1 for integrity) that is used for the connection.

1.2.6.2 Security policy

If security is required, the SP provides *general guidelines* for how it should be provided, and if necessary, links to more specific details. If security is not required, the policy can specify that no security is needed and can disable security for specific packets.

The Linux kernel contains the *Security Policy Database* (SPD) in which all the policies are stored. The Linux kernel decides if a packet should be encrypted, sent without encryption or dropped by consulting the SPD.

The SPD can be managed using the ip xfrm policy command, which will be explained in chapter Command: ip xfrm.

---

6. explained in chapter Security association
5. explained in chapter Security policy

1.2.6.3 Authentication Header

The *AH* is one of two *protocols* that provide traffic security services. AH can offer *integrity* and *data origin authentication*, with optional *anti-replay* features. The protocol also provides access control.

AH provides authentication for as much of the IP header as possible. Some parts of the header may change during transit and therefore cannot be protected by the AH, as the sender cannot predict these changes. An example being the size of the packet in the header, as the value can be changed if a packet is too large and has to be fragmented in order to proceed through the network.

1.2.6.4 Encapsulating Security Payload

The *ESP* protocol provides all of the functionality AH does, with the added benefit of confidentiality.

The ESP header is inserted after the IP header and before the next *layer protocol header* (when in transport mode[1]) or before an *encapsulated IP header* (when in tunnel mode[1]).

1.2.6.5 Transport and Tunneling modes

Two modes can be used with IPsec to protect a packet.

Transport mode only encrypts the payload of the IP packet. The IP header is not modified or encrypted in this mode. Using transport mode with AH results in the inability to use Network address translation (NAT)[2], as translating the address will invalidate the hash value.

Tunneling mode encrypts the entire packet which is then encapsulated into a new packet with a new IP header. Tunnel mode can be used in conjunction with NAT.

1. explained in chapter Transport and Tunneling modes
2. used for remapping the packets IP address

2 Network manipulation with ip

The *ip command* can be used to configure most aspects of the Linux networking stack and is the main component of the iproute2 package. Basic examples of its use are *modifying the IP address* assigned to a network interface, *creating a network tunnel*, *viewing network neighbors*, *creating and managing network routes and rules* and *viewing and managing link layer information* (such as the MAC address of a network interface).

The ip command has several *sub-commands*, each specifically designed to manage a part of the Linux network stack. The following chapters will be explaining each sub-command with a few basic examples of use.

Several commands have *aliases*, which achieve the same functionality but use less characters and are therefore shorter to type by a user. If a shorter version of the command exists, both options will be written.

Although not all commands require root access, it is expected that the reader of this work runs all of the commands as root as not all commands can be fully executed under non-root user privileges.

2.1 Command: ip address

This command is used to manipulate the ip addresses assigned to a specific network interface.

2.1.1 Getting information

Both

```
ip address
ip a
```

commands will print the information about all the network interfaces. An example output follows.

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc  \
```

```
 noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd  \
 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP>  \
 mtu 1500 qdisc pfifo_fast state UNKNOWN qlen 1000
    link/ether 00:1f:d0:95:db:74 \
 brd ff:ff:ff:ff:ff:ff
    inet 192.168.77.25/24 brd 192.168.77.255 \
 scope global eth0
    inet6 fe80::21f:d0ff:fe95:db74/64 scope link
       valid_lft forever preferred_lft forever
3: eth1: <NO-CARRIER,BROADCAST,MULTICAST,UP> \
 mtu 1500 qdisc noop state UP qlen 1000
    link/ether 00:0d:b9:1a:ec:9d \
 brd ff:ff:ff:ff:ff:ff
```

The first line contains basic information about the the *loopback interface,* which has the default loopback IP address 127.0.0.1 assigned.

```
1: lo: <LOOPBACK,UP,LOWER\_UP> mtu 16436 qdisc \
 noqueue state UNKNOWN
```

This line shows that the device is the first network interface and is called lo. The parameters in bracelets (also called *flags*) mean that its a loopback device (LOOPBACK), is UP (the device is running and operational) and has a network cable connected (LOWER_UP). The maximum transmit unit[1] (MTU) is set to 16436 bytes. The next part states which queuing discipline the network interface is using (noqueue). The state of the device is UNKNOWN.

The next line specifies the link type of the device, the MAC address of the device and the broadcast mac address(0:00:00:00:00:00). The next two lines show the *inet (IPv4)* and *inet6 (IPv6)* addresses that are assigned to the interface.

1. the largest size of a packet that is accepted

An interface can have *more than one* inet or inet6 address assigned.

Ip address can be used to view *only IPv4* or IPv6 information about the interface.

```
ip -4 a
ip -6 a
```

The first command will print out only IPv4 information while the second command will print out only IPv6 information.

The *NO-CARRIER* flag on the eth1 interface states that the interface is UP but the network cable is not plugged in, so it cannot function properly.

An *interface* can be specified in the *ip a* command and it will print out information about that interface only. Both following commands are synonymous.

```
ip address show eth0
ip a s eth0
```

They will only print out the information about the eth0 interface.

```
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> \
 mtu 1500 qdisc pfifo_fast state UNKNOWN qlen 1000
    link/ether 00:1f:d0:95:db:74 \
 brd ff:ff:ff:ff:ff:ff
    inet 192.168.77.25/24 brd 192.168.77.255 \
 scope global eth0
    inet6 fe80::21f:d0ff:fe95:db74/64 scope link
       valid_lft forever preferred_lft forever
```

2.1.2 Adding a new address to a network interface

An IP address can be assigned to a network interface by using the

```
ip a a 192.168.77.26/16 brd 192.168.77.255 \
 dev eth0
```

command. This would assign a second IP address to the network interface eth0. The /16 specifies the the *subnet mask* of this address (255.255.0.0). Brd means broadcast followed by an IP address specifying the broadcast address. The state of the device would be changed, as it can be seen in the following output.

```
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> \
 mtu 1500 qdisc pfifo_fast state UNKNOWN qlen 1000
    link/ether 00:1f:d0:95:db:74 \
 brd ff:ff:ff:ff:ff:ff
    inet 192.168.77.25/24 brd 192.168.77.255 \
 scope global eth0
    inet 192.168.77.26/16 brd 192.168.77.255 \
 scope global eth0
    inet6 fe80::21f:d0ff:fe95:db74/64 scope link
       valid_lft forever preferred_lft forever
```

2.1.3 Removing an existing IP address

A previously assigned IP address can be removed from a network interface with the

```
ip a del 192.168.77.25/24 dev eth0
```

command. Checking the state of the device with *ip a s dev eth0* will return:

```
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> \
 mtu 1500 qdisc pfifo_fast state UNKNOWN qlen 1000
    link/ether 00:1f:d0:95:db:74 \
 brd ff:ff:ff:ff:ff:ff
    inet 192.168.77.26/16 brd 192.168.77.255 \
 scope global eth0
    inet6 fe80::21f:d0ff:fe95:db74/64 scope link
       valid_lft forever preferred_lft forever
```

The original inet address was removed and there is only one IPv4 address left on the interface.

2.2 Command: ip addrlabel

Ip addrlabel is used to view and modify the Linux *IPv6 policy table*[1]. It can be used to assign *labels*[2] to IPv6 subnets. Labels can be used to specify the source address of an outgoing packet. Labels with lower numbers take precedence over higher numbered labels.

IPv6 needs a sophisticated method of selecting the source and destination address as a single host can have multiple IP addresses assigned. *RFC3484* specifies a general algorithm that should be used to address this problem.

Each address should have a precedence value and label assigned. *Precedence* selects the destination address and *label* selects the source address. Linux applications assign the precedence value and therefore select the destination IPv6 address. The label value can be manually set using ip addrlabel.

The *label table* contains the list of prefixes that have labels assigned.

Label is a string of at most fifteen characters, not including the NULL terminator. The label allows particular source address prefixes to be used with destination prefixes of the same label. The IPv6 source address selection algorithm prefers a source addresses whose label is equal to the destination address.

2.2.1 Viewing IPv6 labels

```
ip addrl
```

will print out existing labels

```
prefix ::1/128 label 0
prefix ::/96 label 3
prefix ::ffff:0.0.0.0/96 label 4
prefix 2001::/32 label 6
prefix 2001:10::/28 label 7
```

1. longest matching lookup table, used for selecting the source and destination IPv6 address, specified in *RFC3484*[12]
2. also specified in RFC3484

```
prefix 2002::/16 label 2
prefix fc00::/7 label 5
prefix ::/0 label 1
```

Each IPv6 address or subnet can have a label assigned. The label number can be an integer as low as -999999999 and as high as 999999999.

2.2.2 Adding a policy

Ip addrlabel can add a new entry to the Linux IPv6 policy table.

```
ip addrlabel add prefix 2002::2/64 label 99
```

This command will add the prefix 2002::2 with the 64 bit subnet to the policy table with the label value 99.

2.2.3 Removing a policy

Ip addrlabel can also be used to remove existing prefixes with labels assigned to them.

```
ip addrlabel del prefix 2002:102:c0a8:4d01::/64 \
 label 1
```

This command will remove the prefix 2002:102:c0a8:4d01::/64 from the policy table.

2.3 Command: ip link

The ip link command is used for displaying and modifying *link layer information*, such as *enabling promiscuous mode*[1], *disabling multicast* or *changing the MAC address* of a network interface.

1. the whole packet is sent to the CPU instead of just the header, this mode must be enabled if the contents of every network packet processed need to be read

2.3.1 Getting information

The default

```
ip link
ip l
```

commands will show the link information of each network interface.

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 \
 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 \
 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> \
 mtu 1500 qdisc pfifo_fast state UNKNOWN qlen 1000
    link/ether 00:1f:d0:95:db:74 \
 brd ff:ff:ff:ff:ff:ff
```

The output is similar to the output of *ip a* but only shows the link line.

link/ether 00:1f:d0:95:db:74 is the *MAC address* of the network interface, and *brd ff:ff:ff:ff:ff:ff* is the *broadcast MAC address* of the eth0 network interface.

A search that only prints out the interfaces that are UP can be achieved by either of these identical commands

```
ip l sh up
ip link show up
```

As with most of the ip commands, printing out information about a specific device can be achieved by adding the dev parameter.

```
ip link show dev eth1
ip l sh eth1
```

will only print out the link layer information about the eth1 interface.

```
ip -s link
```

will output expanded statistics of all the network interfaces

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 \
 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 \
 brd 00:00:00:00:00:00
    RX: bytes  packets  errors  dropped overrun \
 mcast
    36080      720      0       0       0       0
    TX: bytes  packets  errors  dropped carrier \
 collsns
    36080      720      0       0       0       0
2: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> \
 mtu 1500 qdisc pfifo_fast state UNKNOWN qlen 1000
    link/ether 00:16:17:dc:f2:42 \
 brd ff:ff:ff:ff:ff:ff
    RX: bytes  packets  errors  dropped overrun \
 mcast
    23704208   36167    0       0       0       0
    TX: bytes  packets  errors  dropped carrier \
 collsns
    7863737    40663    0       0       0       0
```

Four lines have been added to each interface that show the sum of *received (RX)* and *transferred (TX)* packets in bytes and the actual number of sent and received packets.

The other values are self explanatory and show the *number of packets affected by errors*, number of *dropped* and *overrun*[1] *packets*, the number of *multicast*[2] packets. The TX line shows the number of *failed carrier*[3] *packets* and the number of *packet collisions*[4].

1. packet length is longer than expected
2. an optimized delivery method of identical packets to several hosts
3. failed to verify if the line is clear before sending any data
4. a collision occurs if two or more hosts attempt to transmit a packet at the same time

The output of the link command can be expanded even more by using a second -s switch.

```
ip -s -s link show dev eth1
```

This command will print out the following output.

```
eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 \
 qdisc pfifo_fast state UNKNOWN qlen 1000
    link/ether 00:16:17:dc:f2:42 \
     brd ff:ff:ff:ff:ff:ff
    RX: bytes  packets  errors  dropped overrun \
   mcast
    23666838   35950    0       0       0       0
    RX errors: length  crc     frame   fifo \
   missed
               0        0       0       0       0
    TX: bytes  packets  errors  dropped carrier \
   collsns
    7802149    40422    0       0       0       0
    TX errors: aborted fifo    window  heartbeat
               0        0       0       0
```

This output is expanded with information abut the number of RX and TX errors that appeared.

length – wrong packet length.

crc – Cyclic redundancy check failed.

frame – A *framing error* is when the network card detects a packet that can't possibly be valid.

missed – the number of missed packets.

aborted – the number of aborted packets.

fifo – Rx FIFO (first-in-first-out) sync errors indicate that the receive FIFOs used by the BIF-RX circuit either got out of sync with the start of frames or with each other. These errors mean that two frames have been discarded. However, those frames did not necessarily contain any user data[27].

window – the number of window size errors.

heartbeat – the number of Signal Quality Errors (SQE)[26].

2.3.2 Controlling the network interface

A network interface can be enabled or disabled with the ip link command. This is useful if a security risk is present and the administrator needs to shut down an interface temporarily to remove the risk or if a network interface needs to be renamed.

```
ip l s eth0 down
```

The above command will put the eth0 interface down. The interface can be re-enabled with the following command.

```
ip l s eth0 up
```

Different aspects of a network interface can be changed.

```
ip link set dev eth0 promisc on
```

will enable promiscuous mode on the Ethernet device eth0. An interface must be in promiscuous mode for a *snooping server*[1] to work.

Disabling promiscuous mode is done by specifying off after promisc instead of on.

```
ip link set dev eth0 promisc off
ip l s eth0 promisc off
```

ARP, dynamic[2], multicast, allmulticast[3], trailers [4] flags can also be enabled or disabled with ip link.

```
ip l set dev eth1 promisc on arp on dynamic \
 on multicast on allmulticast on trailers on
```

1. a snoop server captures network traffic for analysis
2. advisory flag meaning the device is dynamically created and destroyed[9]
3. every multicast packet will be received, regardless of the destination address
4. trailer encapsulation – TCP IP headers are at the end of the packet, less memory to memory copies needed by the system, specified in *RFC 893*[17]

The example above enables all of the aforementioned modes on the eth1 network interface.

```
ip link set eth1 mtu 1400
```

The above example will change the maximum transmit unit value on the eth1 interface to 1400.

The next command will change the transmit queue length of the device to 1000 packets. Both the longer version and its shorter alias are shown.

```
ip link set eth0 txqlen 1000
ip l s eth0 t 1000
```

A lower value is recommended in the case of a slow bandwidth high latency network. A higher value is recommended in the case of a high bandwidth low latency network.

The MAC address of a network interface can be changed by executing any of the two following commands.

```
ip link set dev eth0 address 00:16:17:dc:f2:43
ip l s eth0 a 00:16:17:dc:f2:43
```

This command will only work when the interface is DOWN, as the MAC address cannot be changed while the device is active.

2.3.3 Renaming a network interface

A network interface can be renamed with

```
ip l s eth0 name ethy
```

The network interface will be renamed from eth0 to ethy. Take note that the interface being renamed has to be in the DOWN state, otherwise the command will be refused. You cannot rename an active (UP or UNKNOWN state) interface.

2.4 Command: ip maddr

This command is used to *view all multicast addresses* (IPv4, IPv6 and link layer) assigned to an interface and *add or remove* a *link layer multicast address* that is assigned to an interface.

```
ip maddr
```

will print out all the existing multicast addresses that are assigned to each network interface. It prints out information about interfaces that are either in the UP, DOWN or UNKNOWN state. An example output follows.

```
1: lo
 inet 224.0.0.1
 inet6 ff02::1
2: eth0
 link c0:a8:4d:01:00:00 static
 link 01:00:5e:00:00:01
 inet 224.0.0.1
 inet6 ff02::1 users 2
```

The first device is the loopback interface, which has the multicast IPv4 address 224.0.0.1 and ff02::1 multicast address for IPv6. The second device is the eth0 interface.

The first line of the output shows the *interface index* (1:) and its *name* (lo). Then the multicast address list follows, where each line starts with the *protocol identifier*. The word *link* denotes a *link layer multicast addresses*, while *inet* stands for *IPv4* and *inet6* stands for *IPv6*. The static flag on the second link layer multicast address of the eth0 device means that the address was added with the *ip m add* command. An example of its usage will be shown in chapter Adding and removing a multicast address.

If a multicast address has more than one user, the number of users is shown after the users keyword, as can be seen on the last line of the output after the inet6 address on the interface eth0.

A *specific device* can be selected in the *ip maddr* command to filter the output and make it easier to look trough the information gath-

ered. These two commands are synonymous:

```
ip maddr show dev eth0
ip m l eth0
```

and will both print out multicast information about the eth0 interface only.

2.4.1 Adding and removing a multicast address

The ip maddr add command can be used to add new multicast addresses to a network interface. Taking the existing network setup from the previous chapter, the command

```
ip maddr add 225.0.0.113 dev eth0
```

will add a link layer multicast address to the network interface eth0.

```
ip m l eth0
```

will print out

```
2: eth0
link e1:00:00:71:00:00 static
 link c0:a8:4d:01:00:00 static
 link 01:00:5e:00:00:01
 inet 224.0.0.1
 inet6 ff02::1 users 2
```

with one new line of output present.

The second line *link e1:00:00:71:00:00 static* is the newly added multicast address, the *static flag* at the end meaning it was added with the *ip maddr add* command. Every new multicast address that is added to an interface by *ip maddr* will be flagged as static.

Removing the multicast address is similar. The command

```
ip m del 225.0.0.113 dev eth0
```

will remove the previously added link layer multicast address from the eth0 interface.

2.5 Command: ip neighbor

Ip neighbor (or ip neighbour, its British synonym) can be used to view and manage the Linux *ARP cache*. All of the following commands will use ip neighbor, but ip neighbour can be used in all of the example cases to the same effect.

2.5.1 Viewing the ARP cache

```
ip n s
ip neighbor show
```

will print out the the contents of both the Linux ARP and NDISC cache.

```
acd4::4 dev eth1 lladdr 00:38:d8:25:83:00 \
 PERMANENT
192.168.77.1 dev eth1 lladdr 00:15:6d:be:59:32 \
 REACHABLE
192.168.77.8 dev eth1 lladdr 00:11:d8:55:88:25 \
 STALE
```

The first line shows that the neighbor with the acd4::4 IPv6 address can be accessed through the eth1 interface and has the MAC address 00:38:d8:25:83:00. Permanent means that the NDISC cache entry was added manually (with *ip neighbor add*, an example will be shown in chapter Adding a new neighbor) and does not expire like other automatically generated cache entries do.

The second line show that the neighbor with the IP address 192.168.77.1 is accessed through the eth1 interface, has the link layer address 00:15:6d:be:59:3200:15:6d:be:59:32 and is currently reachable.

The third line show that the neighbor with the IP address 192.168.77.8 is in a stale state, meaning that it is still usable but needs

verification.

Other possible states are *permanent* (meaning the arp cache entry never expires), *noarp* (meaning the entry has normal expiration but was never verified), *incomplete* (the first ARP request was sent), *delay* (ARP request was scheduled), *failed* (no ARP response was received), *probe* (sending ARP request) and *none*.

2.5.2 Adding a new neighbor

Ip neighbor add can be used to add a new arp table entry and bypass the automatic arp table generation.

```
ip n add 192.168.77.23 lladdr 00:c0:7b:7d:00:c8 /
 dev eth1 nud perm
```

will add a new network neighbor with the IP address 192.168.77.23 and link layer address 00:c0:7b:7d:00:c8. This neighbor will be permanently present in the ARP cache and will never expire. This can be used to disable access to a specific host, as if an incorrect link layer address is specified no connection can be established to the host.

2.5.3 Removing a neighbor

Ip neighbor del can be used to remove an existing ARP cache entry.

```
ip n del 192.168.77.1 dev eth1
```

will remove the ARP cache entry for the 192.168.77.1 address. Verifying the change with *ip n* will return the following output.

```
192.168.77.1 dev eth1  FAILED
192.168.77.8 dev eth1 lladdr 00:11:d8:55:88:25 \
 STALE
```

meaning that 192.168.77.1 has no link layer address assigned. Removing the ARP cache entry won't disable access to the 192.168.77.1 host as a new entry will be regenerated when the host is accessed again provided that ARP is enabled on that host.

2.6 Command: ip ntable

This command is used to print information about the Linux kernel ARP and NDISC cache.

2.6.1 Viewing the ARP and NDISC cache information

The basic

```
ip ntable
```

command will print out ARP and NDISC cache information about every Linux network interface. A specific device or ntable name can be selected and ip ntable will print information about that device only. The following two synonymous commands

```
ip ntable show dev eth0
ip nt s dev eth0
```

will print out

```
inet6 ndisc_cache
    dev eth0
    refcnt 1 reachable 33980 \
       base_reachable 30000 retrans 1000
    gc_stale 60000 delay_probe 5000 queue 3
    app_probes 0 ucast_probes 3 mcast_probes 3
    anycast_delay 1000 proxy_delay 800 \
     proxy_queue 64 locktime 0

inet arp_cache
    dev eth0r
    refcnt 4 reachable 20070 \
       base_reachable 30000 retrans 1000
    gc_staple 60000 delay_probe 5000 queue 3
    app_probes 0 ucast_probes 3 mcast_probes 3
    anycast_delay 1000 proxy_delay \
 800 proxy_queue 64 locktime 1000
```

The first paragraph *inet6 ndisc_cache* shows all the available information about the IPv6 NDISC cache of the eth0 network interface.

Refcnt shows the number of references the NDISC cache has. Values above 0 prevent the NDISC cache to be prematurely freed.

Reachable – once a neighbor has been found, the neighbor entry is considered to be valid for at least a random value that is between $base\_reachable\_time/2$ and $3 * base\_reachable\_time/2$. The result of this calculation is the reachable value (in milliseconds).

Base_reachable shows the approximate length of time (in milliseconds) that the device maintains reachable status for a neighbor after the security device transmits a *Neighbor Solicitation* message to the neighbor and receives a *Neighbor Answer* message in reply.

Retrans is the time (milliseconds) between retransmitted Neighbor Solicitation messages. Used for address resolution and to determine if a neighbor is unreachable.

Gc_stale determines how often to check for stale ARP entries (in milliseconds). After an ARP entry is stale it will be resolved again.

Delay_probe is the delay for the first time probe if the neighbor is reachable (in milliseconds).

Queue is the length of the ARP queue, all packets to be sent for neighbor entries in the *incomplete state* are collected here. *The queue size is in packets.*

App_probes shows the number of solicitations that can be sent by a user-space application when resolving an address.

Ucast_probes is the number of unicast solicitations that can be sent to confirm the reachability of an address.

Mcast_probes is the number of multicast solicitations that can be sent to resolve a neighbors address. For ARP this is the number of broadcast solicitations, because ARP does not use multicast solicitations, but IPv6 does. Mcast and app probes are mutually exclusive, only one can be a non-zero value.

Anycast_delay is the maximum for random delay of answers to neighbor solicitation messages (in milliseconds).

Proxy_delay is the maximum time before answering to an ARP request for which there is a proxy ARP entry. In some cases, this is used to prevent network flooding.

Proxy_queue is the maximum queue length of the delayed proxy ARP timer.

Locktime – An ARP entry is only replaced with a new one if the old is at least locktime old. This prevents ARP cache thrashing. Value is in milliseconds.

More information about ARP can be read on the ARP module manual page accessed by the *man 7 arp* command [3] [4]

The output of the ip ntable command can be limited to one type of cache.

```
ip ntable show name arp_cache
ip nt s name arp_cache

ip ntable show name ndisc_cache
ip nt s name ndisc_cache
```

The first two commands will print out information about all ARP caches only while the second pair will return information only about all NDISC caches. In addition to the caches that are assigned to each interface, these commands will print out the main ARP and NDISC settings that have information about cache garbage collection added.

This is an example of the global cache settings:

```
inet arp_cache
    thresh1 128 thresh2 512 thresh3 \
        1024 gc_int 30000
    refcnt 1 reachable 35993 base_reachable \
        30000 retrans 1000
    gc_stale 60000 delay_probe 5000 queue 3
    app_probes 0 ucast_probes 3 mcast_probes 3
    anycast_delay 1000 proxy_delay 800 \
        proxy_queue 64 locktime 1000
```

The information present on the second line is new and contains variables that affect cache garbage collection. The ndisc_cache table has the same variables, adding extra lines of the same output (ip ntable show name ndisc_cache) is therefore unnecessary.

Tresh1 specifies the bottom size limit of the ARP/NDISC cache, no garbage collection is done unless the number of cache entries exceeds this value. Value means the number of cache entries. The tresh1 value was used in the 2.4.x kernels and is not used anymore in the 2.6.x kernels, but the value is still present in the source code.

Tresh2 is the ARP/NDISC cache value that should be exceeded only briefly.

Tresh3 is the absolute upper limit size of the ARP/NDISC table that must not be exceeded.

If tresh2 cache size is achieved, garbage collection will start until the number of ARP entries is below tresh1.

Gc_int specifies how frequently the garbage collector for neighbour entries should attempt to run (in milliseconds). Defaults to 30 seconds.

2.6.2 Modifying the ARP and NDISC cache information

Ip ntable can also be used to modify any of the values it prints information about.

The syntax of the command is *ip ntable change [dev DEVname](optional) name cacheNAME(arp_cache or ndisc_cache, required) VARIABLE variableVALUE (variable being any of the parameters)*.

The following command will change the retrans value of the ARP cache on device eth1 to 3000.

```
ip ntable change dev eth1 retrans 3000 \
 name arp_cache
```

If no device is specified with the dev parameter, the main ARP or NDISC variables will be changed.

```
ip ntable change retrans 1000 name arp_cache
```

The main ARP and NDISC variable values are applied to any new network interfaces that are created (for example with the ip tunnel add command).

2.7 Command: ip route

The *ip route* command can view and edit the Linux routing tables.

Routing tables on Linux are numbered from 0 to 255, where 254 is the *main table* and 255 is the *local table*. The routing table names are store in the */etc/iproute2/rt_tables* file and more tables can be added to the file if needed.

2.7.1 Getting information

Simply typing

```
ip r
```

will show the contents of the main routing table.

The number (0 to 255) or name of an existing routing table can be added to the command. Therefore both following command will print out the contents of the main (254) table.

```
ip route show table main
ip r s t 254
```

The output follows.

```
192.168.77.0/24 dev eth1  proto kernel  \
   scope link  src 192.168.77.24  metric 1
default via 192.168.77.1 dev eth1  proto static
```

Routing information assigned to a specific address can be viewed by specifying the address.

```
ip route get 192.168.77.77
ip r g 192.168.77.77
```

If the IP address is located on a local network behind a gateway, the ip r output won't list the gateway address as it is not needed to pass. If the address is outside the local network behind a gateway, the output will list that the route leads through a gateway. An example

follows.

There are 3 network addresses: 192.168.77.77, 192.167.77.1 and 194.1.130.87. The first two are located on the same subnet, the second being the gateway of that subnet. Ip r get 192.168.77.77 gives the output

```
192.168.77.77 dev eth0  src 192.168.77.25
    cache  mtu 1500 advmss 1460 hoplimit 64
```

meaning it did not need to pass the gateway. However, the command

```
ip r get 194.1.130.87
```

returns the output

```
194.1.130.87 via 192.168.77.1 dev eth0 \
  src 192.168.77.25
    cache  mtu 1500 advmss 1460 hoplimit 64
```

meaning it had to pass the gateway 192.168.77.1 to get to the address.

The -s parameter can be added to the ip route command to show expanded information about the routes.

```
ip -s r show cache 192.168.77.1
```

will return

```
192.168.77.1 dev eth0  src 192.168.77.25
    cache  users 1 used 4 age 11sec mtu 1500 \
advmss 1460 hoplimit 64
192.168.77.1 from 192.168.77.25 dev eth0
    cache  users 1 used 3 age 11sec mtu 1500 \
advmss 1460 hoplimit 64
```

The expanded information shows the *number of users using the route* (users 1), the number of *times the route has been used* (used 4) and the *age of the routing cache entry* (age 11sec) in addition to the maximum transmit unit (mtu), advertised maximum segment size (advmss) and hoplimit.

Advmss is the *maximum size of the payload*, and is calculated by subtracting 40 bytes from the MTU value (20 bytes for the TCP header and 20 bytes for the IPv4 header)[16].

2.7.2 Viewing the Linux routing cache

Ip route can also be used to view all of the routes currently stored in the Linux routing cache. The commands

```
ip r show cache
ip r s c
```

will print out a list of all the cached IP addresses.

```
local 192.168.77.25 from 205.188.5.223 dev lo \
 src 192.168.77.25
    cache <local>  iif eth0
broadcast 192.168.77.255 from 192.168.77.8 \
  dev lo src 192.168.77.25
    cache <local,brd,src-direct>  iif eth0
local 192.168.77.25 from 209.85.135.100 dev lo \
 src 192.168.77.25
    cache <local>  iif eth0
193.62.22.98 via 192.168.77.1 dev eth0 \
 src 192.168.77.25
    cache  mtu 1500 advmss 1460 hoplimit 64
...(the list continues and is a few pages long)
```

It is faster for the Linux kernel to lookup and IP address in the cache than to do a lookup each time it needs to find the address. The cache is stored as a hash table.

A single IP address can also be specified when viewing the cache. The command

```
ip r show cache 140.211.166.4
```

will show only the route to the address 140.211.166.4

```
140.211.166.4 from 192.168.77.25 \
  via 192.168.77.1 dev eth0
    cache  mtu 1500 advmss 1460 hoplimit 64
```

All entries of the routing cache can be removed by

```
ip r flush cache
```

Do not use this command in a working environment, as this command can and will disable all networking capability of the Linux kernel when flushing all the entries in the cache with ip route flush all, as it will even disable link layer routing to all the network interfaces on the system.

2.7.3 Managing the routing tables

The ip route command can also be used to add or remove routes to the Linux routing tables. The command

```
ip r a 194.1.130.0/24 via 192.168.77.1
```

adds a routing table entry that redirects all the packets with the destination network of 194.1.130.0/24 to the router with the ip address 192.168.77.1.

The command

```
ip route add prohibit 192.168.77.77
```

will disable access to that IP address, pinging the network address will return *connect: Network is unreachable*. A specific network address can be prohibited to access another address by adding prohibit to the command.

```
ip route add prohibit 192.168.77.77 \
 from 192.168.77.25
```

will block the access to 192.168.77.77 if the source address of the request is 192.168.77.25.

Those rules can be removed by issuing the command

```
ip route del prohibit 192.168.77.77.
```

A default gateway can be added to the Linux routing tables using

```
ip route add default via 192.168.1.254.
```

The default gateway value can also be changed with

```
ip route change default via 192.168.77.1
```

A black hole can also be added to the routing tables.

```
ip route add blackhole 192.168.77.77
```

Adding a black hole will make the destination unreachable.

Possible route types are *unicast*, *local*, *broadcast*, *multicast*, *throw*, *unreachable*, *prohibit*, *blackhole* and *nat* (no longer supported in 2.6.x kernels).

2.7.4 Command: IPv6 and ip route

Ip route can also be used to add and remove IPv6 routes. The command syntax is the same, with the exception that IPv6 addressees are used instead of IPv4. The -6 switch can be used with the ip route command to specify that an IPv6 route is being managed, but it is not normally needed as ip route can determine if the IP address is IPv4 or IPv6 without the need for additional parameters. The ip route commands are therefore identical and only a few examples will be given.

```
ip -6 r a 2002::a via 2002::b
ip r a 2002::a via 2002::b
```

```
ip -6 r d 2002::a via 2002::b
ip r d 2002::a via 2002::b
```

The first two commands will add the 2002::a via 2002::b IPv6 route and the second two will remove that route from the routing tables.

Adding a default IPv6 route is similar.

```
ip -6 r a default via fe80::c0a8:4d18 dev eth2
ip r a default via fe80::c0a8:4d18 dev eth2
```

These two identical commands will create a default IPv6 route that will pass through the eth2 network interface with the IPv6 address fe80::c0a8:4d18.

2.7.5 Using realms with ip route

Realms can be used on systems that need more complex routing tables. Each ip route rule can have a realm assigned as an optional parameter.

Assigning a realm yields two main benefits – the network administrator gains more insight on the network while network management becomes easier.

It is easier to split a larger network into several smaller realms and manage those smaller realms with rules instead of managing the whole network at once. Network usage statistics can also be gained from specified realms with rtacct, so it makes analyzing the network and finding the problem easier for the administrator. A default Linux machine contains only the realm 0 that is named cosmos and is used as the default realm when there is no realm specified or the specified realm can not be found.

The */etc/iproute2/rt_realms* text file contains the list of currently usable realms.

```
#
# reserved values
#
0    cosmos
```

```
#
# local
#
#1    inr.ac
#2    inr.ruhep
#3    freenet
#4    radio-msu
#5    russia
#6    internet
```

This file can be manually edited and *up to 255* additional realms can be specified. Each realm must have a number assigned to it. When specifying a realm with ip route, either the realm name or number can be used.

When a new realm is added to the file, it can be immediately used without the need to restart any networking interface or the networking service itself.

A new realm can be added by opening the file in a text editor and adding a new line or by a simple use of the echo command.

```
echo 10 muni >> /etc/iproute2/rt_realms
```

This command added the muni realm with the number 10 assigned to it to the last line of the rt_realms file.

The newly added realm can be used straight away and a new routing rule that uses the realm can be added.

```
ip r a 192.168.77.0/24 via 192.168.77.1 realm muni
```

Other operations with the ip route command are the same as the examples provided in the other ip route chapters, only the realm parameter needs to be added.

Another useful use of realms is that a realm can be specified when using the *rtacct* network statistic tool that is also part of iproute2. When a realm is specified, rtacct will show the statistics of that realm only.

The command

```
rtacct cosmos
```

will limit rtacct to show the statistics of packets that only belong to the cosmos realm.

2.8 Command: ip rule

Ip rule is used to view, add and remove rules to the Linux routing policy database (RPDB). By using the RPDB, the administrator can select which routing table is to be used with a specific subnet.

Either of the following commands

```
ip rule
ip ru
```

will print out the default routing policy database.

```
0: from all lookup local
32766: from all lookup main
32767: from all lookup default
```

There are three default rules present in the Linux RPDB on every host. The *first rule* (0) states that the local routing table should be used on all packets.

The *second and third* rules specify the use of main and default routing tables if the local or any other table can't be used. The high numbers mean that those two rules will be usually used as the last option, unless a rule with a higher number is manually added.

The RPDB controls the order in which the kernel searches through the routing tables. Each rule has a priority, and rules are examined sequentially from rule 0 through rule 32767.

When a new packet arrives for routing (assuming the routing cache is empty), the kernel begins at the highest priority rule in the RPDB (rule 0). The kernel iterates over each rule in turn until the packet to be routed matches a rule. When this happens the kernel follows the instructions in that rule. This usually causes the kernel to perform a route lookup in a specified routing table. If a matching

route is found in the routing table, the kernel uses that route. If no such route is found, the kernel returns to traverse the RPDB again, until every option has been exhausted.[8]

2.8.1 Adding a new RPDB rule

Several kinds of rules can be added to the RPDB, both simple and more complex. The following command

```
ip rule add from 10.10.10.0/24 pref 32768 \
 table main
```

will add a new rule with the priority of 32768, meaning it will be looked up as the last rule when the kernel is searching the RPDB. This rule will apply the main routing table to any ip packet arriving from the 10.10.10.0/24 subnet.

More complex rules can be added to the RPDB, as can be seen with the following command.

```
ip rule add iif eth0 oif eth1 from 10.10.10.0/24 \
to 10.10.10.0/18 fwmark 1/12 pref 32768\
table main prohibit realms cosmos/cosmos \
type blackhole
```

This rule will be applied to packets incoming on eth0 and outgoing on the eth1 interface that are originating from the 10.10.10.10/24 subnet and are destined for the 10.10.10.0/18 subnet.

Fwmark states that only packets that are marked 1 by the firewall and are using the mask 12 will use this rule, *pref* states that the rule will be looked up by the kernel after all the rules have been searched (unless there is a rule with a higher pref present) and that all packets using this rule will use the main table but also be *prohibited* from being sent. This rule will also only be limited to the outgoing and incoming cosmos realm (cosmos/cosmos). The last parameter states that all packets that match this rule should be dropped and the rule will therefore create a network black hole.

This command is probably *unusable* in a real world application, but shows the power of control over the network that the adminis-

trator can gain by managing the RPDB by the ip rule command.

2.8.2 Removing an existing RPDB rule

Deleting a rule is similar to adding a rule.

```
ip rule del pref 32786
ip r d pref 32786
```

This will remove the first rule in the RPDB that has the preference number 32786.

It is important to note that *several rules* can have the *same pref priority number*. If one rule with the number already exists, a new rule with the same preference number is added after the last rule with that number. This means that the RPDB can have several rules with the same priorities, and the rule that was added first will have precedence over a rule that was added later as the kernel looks through the RPDB sequentially.

The same applies for deleting a rule, the first and only the fist rule with the preference number will always be deleted when only the number of the rule is specified and that number is used by several rules.

A specific rule with the same preference can be deleted out of sequence if all of the parameters present in the rule are specified when deleting, as those will only match the single rule.

2.9 Command: ip tunnel

The ip tunnel command is used for the creation and management of network tunnels on Linux.

2.9.1 IPv6 over IPv4 tunnels

Two IPv6 networks can be connected through an IPv4 network using IPv6 over IPv4 tunnels. The Linux kernel can contain several tunneling protocols that can achieve this. Two most commonly used are *Simple Internet Transition (SIT)* and *Generic Routing Encapsulation*

(GRE), therefore only they will be mentioned and only their example of use will be given.

2.9.2 Simple Internet Transition (SIT)

SIT is a tunneling protocol that can payload IPv6 packets over IPv4. Its main purpose is to interconnect isolated IPv6 networks that are located in a global IPv4 Internet. SIT works like *IPIP*[1].

The kernel module *ipv6.ko* is required for SIT to be available. It is located in */lib/modules/'uname -r'/kernel/net/ipv6/ipv6.ko*

2.9.3 Generic Routing Encapsulation (GRE)

GRE[1] is a tunneling protocol that can payload *different types of packets* (compared to IPv6 packets only for SIT) through an IPv4 network.

GRE is better suited if one of the tunnel endpoints is a Cisco router, as it was developed by *Cisco Systems*[2] and is thus natively used on their hardware.

The */lib/modules/'uname -r'/kernel/net/ipv4/ip_gre.ko* kernel module must be present and loaded for GRE to be available.

2.9.4 IPv6 rapid deployment (6rd)

A new type of *6to4 tunnel* was designed and implemented by *Free*[3] to simplify the transition to IPv6, called *IPv6 rapid deployment* or *6rd* in short.

This tunnel is similar to 6to4, but manages only traffic that is assigned to a specific *Internet service provider (ISP)*. Each ISP is assigned a 32 bit IPv6 prefix that uniquely identifies its own network (compared to a 16 bit 6to4 prefix that identifies 6to4 tunnels). The IPv6 prefix is used in conjunction with the IPv4 address of a specific router on the ISP network.

---

1. Simple tunneling that can encapsulate IPv4 packets only
1. specified in *RFC2784*[28]
2. multinational corporation that designs and sells consumer electronics, networking and communications technology and services
3. French Internet service provider

The first 32 bits of the IPv6 address that identifies the ISP are followed by a 32 bit address of the IPv4 router. If the ISP has the subdomain 2002:102::/32 registered with 6rd and the router has the 192.168.77.1 IP address the resulting IPv6 address of the router will be 2002:102:c0a8:4d01::/64[23].

The hosts behind the router can have a local IPv4 address that will be transformed into an unique IPv6 address using the ISP prefix followed by the 32 bit local IPv4 address. This will assure that every host can have it's own unique IPv6 IP address and can create an easy way for an ISP to slowly migrate from IPv4 to IPv6. 6rd was integrated into the *2.6.33 Linux kernel* and can be managed with the iproute2-ss100224 package.

6rd is not available on a default Fedora 13 installation, Fedora has to be updated with the *yum update* command to update iproute2 and the kernel to 2.6.33 or higher. Updating will enable 6rd configuration with iproute2.

2.9.5 Creating a SIT tunnel

A SIT tunnel can be created by setting up two tunneling network interfaces, each at one end of the tunnel. There are two hosts A and B, A has the IP address 192.168.77.25 and B has 192.168.77.87.

A tunnel network interface on machine A has to be created

```
ip tunnel add tuna mode sit remote 192.168.77.87 \
 local 192.168.77.25
```

Remote stands for the remote address of the second tunnel on host B and local stands for the ip address of host A.

The tunnel interface then has to be set up.

```
ip l set dev tuna up
```

An IPv6 network address is then needed to be added to the interface.

```
ip a add ::a dev tuna
```

This end of the tunnel is now fully set.

These steps need to be repeated on machine B. Take note of the switched remote and local IP addresses.

```
ip tunnel add tunb mode sit local 192.168.77.87 \
 remote 192.168.77.25
ip l set dev tunb up
ip a add ::b dev tunb
```

With both tunnel endpoints on host A and B running, the connection can be tested by pinging the IPv6 address of one interface from the other. Pinging host A from host B with

```
ping6 ::a
```

should return a number of positive replies. The same test should work when pinging B from A.

```
ping6 ::b
```

Successful pings on both sides mean that the tunnels are working. If pings are failing, one of the possible causes could be that IPv6 routing is not properly set, in which case the reader should re-read the *IPv6 and ip route* chapter and create a default IPv6 route that will use the tunnel.

2.9.6 Creating a GRE tunnel

In order for a GRE tunnel to work, its kernel module has to be installed. It is not installed on Linux distributions such as Fedora 13 or Ubuntu 10.04.

If the module is missing, trying to create a GRE tunnel will return the error *ioctl: No such device*.

The needed kernel module can be installed with *insmod*[1]:

```
insmod \
 /lib/modules/`uname -r`/kernel/net/ipv4/ip_gre.ko
```

1. install loadable kernel module, a Linux terminal command

Creating the GRE tunnel is almost identical to a SIT tunnel:

```
ip t add tuna mode gre remote 192.168.77.87 \
 local 192.168.77.25
ip l set dev tuna up
```

An unique IPv4 address can be assigned to the GRE tunnel network interface, one that is outside of pinging range.

```
ip a add 10.0.0.2/24 dev tuna
```

A routing rule that will redirect all the network traffic through the new tunneling interface also needs to be created.

```
ip r a 10.0.0.3 via 10.0.0.2
```

The same steps need to be repeated on the other endpoint of the tunnel with the slight modification of switching the remote and local addresses.

```
ip t add tunb mode gre remote 192.168.77.25 \
 local 192.168.77.87
ip l set dev tunb up
```

A new IP address needs to be assigned to the tunnel, the same address as was added to the routing rule on the first machine.

```
ip a add 10.0.0.3/24 dev tunb
ip r a 10.0.0.2 via 10.0.0.3
```

Testing the tunnel can be done by pinging the 10.0.0.3 IP address. A successful ping means the tunnel is working. Further testing can be done by changing the routing rule and checking that the ping will fail as it cannot find the route to the address without using the tunnel.

The tunneling interface can also be brought down using *ip l set dev tunb down*, but that method of testing is not advised as the tunnel will *forget its settings* (assigned IP addresses) when putting it up again

and all the necessary settings will have to be added again.

2.9.7 Creating a 6rd tunnel

Fedora 13 was used in this example, but any Linux distribution with kernel 2.6.33 or higher can also be used.

For Fedora 13 to gain 6rd functionality, it has to be updated with the *yum update* command. After updating iproute2 and the Linux kernel, a *6rd tunneling interface* can be created and configured with the ip tunnel command. The following 6rd tunneling mode commands were first published by *Nathan Lutchansky*[20].

```
ip tunnel add 6rdtun mode sit local 10.10.10.10 \
 ttl 64
ip tunnel 6rd dev 6rdtun 6rd-prefix 2001:55c::/32
ip link set 6rdtun up
```

The first command will create a sit tunnel named 6rd tun with the local IP address 10.10.10.10. The second command will assign a 6rd prefix to the tunneling interface, the prefix of an ISP that provides 6rd. The third command will set the tunneling interface up.

2.10 Command: ip xfrm

The *ip xfrm* command can access the Linux *XFRM (transform) framework*, which can be used to manipulate incoming and outgoing packets. XFRM can be used to extend IPsec functionality to active connections and thus authenticate or secure traffic.

There are two sub-commands of ip xfrm – *policy* and *state*. Policy is used to access the *Security Policy Database (SPD)* while state is used to access the *Security Association Database (SAD)*. An IPsec connection must have both SAD and SPD entry in order to work.

2.10.1 Managing the SAD

Ip xfrm can be used to create a new SA and add it to the SAD.

```
ip xfrm state add src 192.168.77.23 dst \
 192.168.77.24 proto esp spi 0x53fa0fdd \
 mode tunnel reqid 16386 replay-window 32 \
 auth "hmac(sha1)" \
 0x55f01ac07e15e437115dde0aedd18a822ba9f81e \
 enc "cbc(aes)" \
 0x6aed4975adf006d65c76f63923a6265b sel \
 src 0.0.0.0/0 dst 0.0.0.0/0
```

This command will add a new SA to the SAD that affects incoming traffic from 192.168.77.23 that is destined to 192.168.77.24. The SA protects the *uni-directional* traffic by using the ESP protocol.

Src is the source IP address, *dst* is the destination IP address, *proto* defines the type of protocol used (ESP in this example) with a specific *Security Parameters Index*[1] (SPI) (0x53fa0fdd) and the type of packet *transport mode (tunneling)*.

Reqid of the SA and SP have to be identical, otherwise IPsec communication will not work and all packets will be dropped.

Replay-window specifies the number of packets the decrypter can keep track of.

Auth specifies the type of authentication algorithm used (sha1) followed by the value of the key.

Enc specifies the type of the encryption algorithm used CBS (aes) followed by a key.

Sel selects the *source (src)* and *destination (dst)* address range of the SA.

The newly added SA can be viewed using either of the following commands.

```
ip xfrm state
ip x s
```

An existing SA can be removed with

```
ip xfrm state delete src 192.168.77.23 \
```

1. Arbitrary value which is used to identify the security association of the sending party

```
dst 192.168.77.24 proto esp spi 0x53fa0fdd
```

This command deleted the SA that was added at the beginning of this chapter. At least the source address, destination address and protocol have to be specified with the delete command.

Every SA in the SAD can be be removed with the flush command.

```
ip xfrm state flush
ip x s f
```

2.10.2 Managing the SPD

After creating SA, SP has to be created that will filter all desired traffic through the SA and therefore encrypt it.

A policy can be added to the SPD with

```
ip xfrm policy add dir in src 192.168.1.20/32 \
 dst 192.168.1.6/32 ptype main action allow \
 priority 2080 tmpl src 192.168.1.20 \
 dst 192.168.1.21 proto esp reqid 16386 \
 mode tunnel
```

Updating a policy is very similar

```
ip xfrm policy update dir in src 192.168.1.6/32 \
 dst 192.168.1.21/32 ptype main action \
 allow priority 2080 tmpl \
 src 192.168.1.20 dst 192.168.1.21 \
 proto esp spi 0x50a0e242 mode tunnel \
 reqid 16386
```

When deleting a policy, only the source address, destination address and type have to be specified.

```
ip xfrm policy delete dir in src 192.168.77.23 \
 dst 192.168.77.24 ptype main
```

Every policy can be removed using either of the following commands.

```
ip xfrm policy flush
ip x p f
```

2.10.3 Working IPsec example

This chapter will contain a simple example of using ip xfrm to set up an IPsec connection between two IPsec capable hosts.

There are two hosts, *192.168.77.23* and *192.168.77.24*. As the IPsec connection is *full-duplex*, *two SA* and *SP* have to be added on each host. This set of commands will add the desired SA and SP on the 192.168.77.23 host.

```
ip xfrm state add src 192.168.77.23 \
dst 192.168.77.24 proto esp spi 0x53fa0fdd \
mode transport reqid 16386 replay-window 32 \
auth "hmac(sha1)" \
0x55f01ac07e15e437115dde0aedd18a822ba9f81e \
enc "cbc(aes)" \
0x6aed4975adf006d65c76f63923a6265b \
sel src 192.168.77.23 dst 192.168.77.24; ip xfrm \
state add src 192.168.77.24 dst 192.168.77.23 \
proto esp spi 0x53fa0fdd mode transport reqid \
16386 replay-window 32 auth "hmac(sha1)" \
0x55f01ac07e15e437115dde0aedd18a822ba9f81e \
enc "cbc(aes)" \
0x6aed4975adf006d65c76f63923a6265b sel src \
192.168.77.24 dst 192.168.77.23; ip xfrm \
policy add dir out src 192.168.77.24 dst \
192.168.77.23 ptype main action allow priority \
2080 tmpl src 192.168.77.24 dst 192.168.77.23 \
proto esp reqid 16386 mode transport; ip xfrm \
policy add dir in src 192.168.77.23 dst \
192.168.77.24 ptype main action allow priority \
2080 tmpl src 192.168.77.23 dst 192.168.77.24 \
```

```
proto esp reqid 16386 mode transport
```

The second set of commands will add the same set of SA and SP on the 192.168.77.24 host. Notice that the *in* and *out* addresses have been reversed in the case of SP but the SA did not need such a change, as they are configured for each way once. The *reqid* parameter *has to be the same* for SA and SP otherwise the host will drop all packets as it will not know which SP to associate to which SA.

```
ip xfrm state add src 192.168.77.23 \
dst 192.168.77.24 proto esp spi 0x53fa0fdd \
mode transport reqid 16386 replay-window 32 \
auth "hmac(sha1)" \
0x55f01ac07e15e437115dde0aedd18a822ba9f81e \
enc "cbc(aes)" \
0x6aed4975adf006d65c76f63923a6265b sel src \
192.168.77.23 dst 192.168.77.24; ip xfrm \
state add src 192.168.77.24 dst 192.168.77.23 \
proto esp spi 0x53fa0fdd mode transport reqid \
16386 replay-window 32 auth "hmac(sha1)" \
0x55f01ac07e15e437115dde0aedd18a822ba9f81e enc \
"cbc(aes)" 0x6aed4975adf006d65c76f63923a6265b \
sel src 192.168.77.24 dst 192.168.77.23; ip \
xfrm policy add dir in src 192.168.77.24 dst \
192.168.77.23 ptype main action allow priority \
2080 tmpl src 192.168.77.24 dst 192.168.77.23 \
proto esp reqid 16386 mode transport; ip xfrm \
policy add dir out src 192.168.77.23 dst \
192.168.77.24 ptype main action allow priority \
2080 tmpl src 192.168.77.23 dst 192.168.77.24 \
proto esp reqid 16386 mode transport
```

After configuring IPsec on both hosts, simply *pinging* the first host from the second and vice-versa will enable IPsec. Pinging from both sides is required, as without both pings the hosts will not be able to communicate.

It's also important to remember that *after rebooting* a host, its IPsec configuration *will be lost*. It is therefore advised to create a *start-up*

script that sets up IPsec each time the host boots.

3 Iproute network shaping

The iproute2 package contains the *tc* tool, which is meant for *traffic control*. Traffic control on Linux can be achieved by using *queuing disciplines (qdiscs)* that are inside the kernel. Qdiscs are *schedulers* that *arrange packets*.

There are *two kinds* of qdiscs on Linux, *classless* and *classfull*. Classless qdiscs can only have a *single scheduler assigned* to them and cannot contain more classes.

Classless qdiscs can be used on a network interface as the primary qdisc or can be used inside a leaf class of a classfull qdisc.

A *classfull* qdisc can have several classes and schedulers assigned and can be used to *shape network traffic*.

3.1 Classless qdiscs

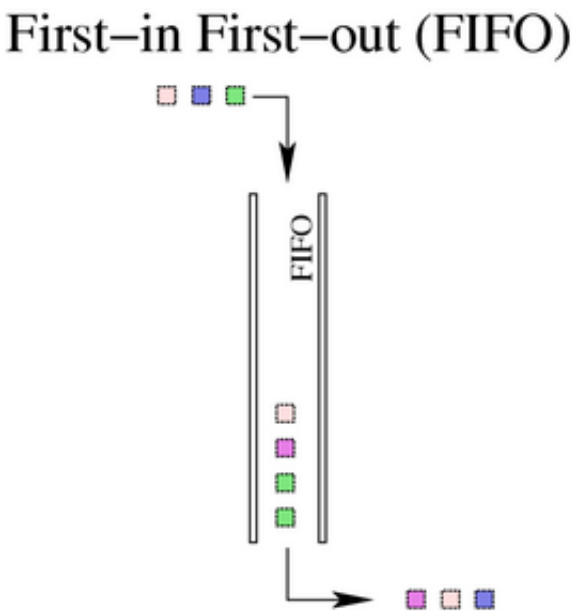
There are several *classless qdiscs* present in the Linux kernel that can be assigned to an network interface. By selecting a qdisc we can control how the computer schedules incoming and outgoing traffic.

There can only be *one* classless qdisc assigned to an interface as a primary qdisc at any given time.

3.1.1 Pfifo and Bfifo

The *Pfifo* and *Bfifo* qdiscs are standard First in, First out *queues*. The difference between them is that Pfifo limits the *size* of a queue by *packets*, while Bfifo does so by *bytes*.

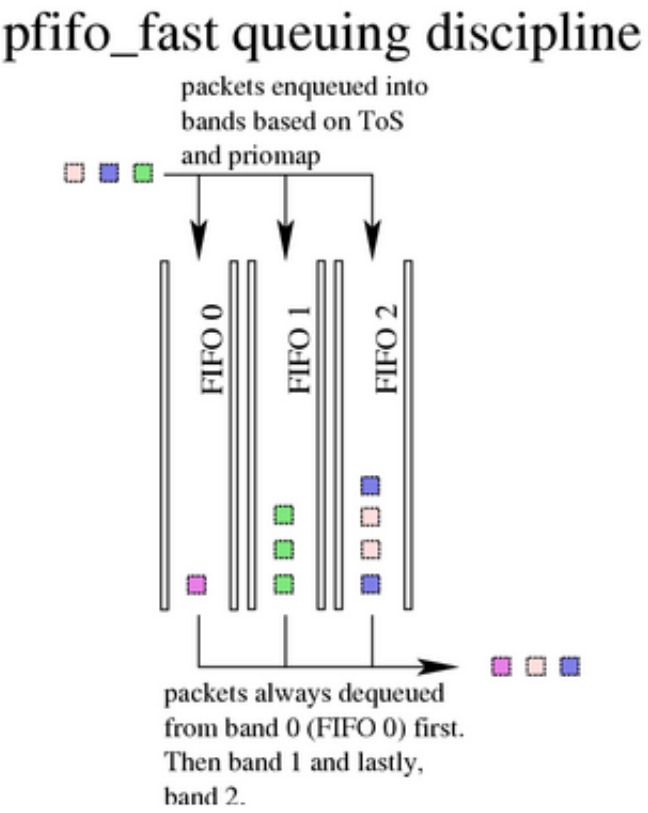
They both also *maintain statistics*, while the default *pfifo_fast* qdisc does *not*.



[7]

3.1.2 Pfifo_fast

This is the *default qdisc* that is used on every *new* network interface. This qdiscs consists of 3 *Fifo queues (bands)* that work side by side. The three Fifo bands are numbered from *0 to 2*, where 0 gets the packets with the *highest priority*, 1 gets the packet with the *medium* priority and 2 gets the packets with the *lowest* priority.



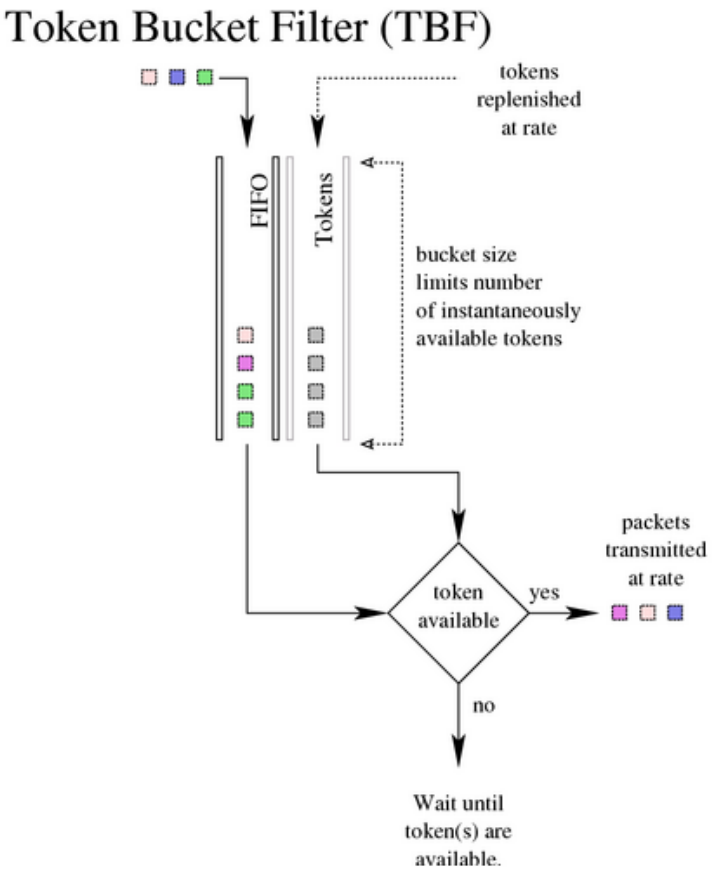
[7]

The bands are emptied *from 0 to 2*, meaning if the 0 buffer is not empty the following buffers wont begin emptying before the first one is emptied. The number of bands and the *priority map* (which packet is assigned to which band) can be changed.

3.1.3 TBF

The *Token Bucket Filter (TBF)* is a qdisc that is used when *simple network shaping* is needed on a single network interface.

It can be summarized as *fifo with tokens* added. By using tokens, the speed at which the interface transmits and receives data can be manipulated. TBF lets packets pass by when there are tokens in the buffer.



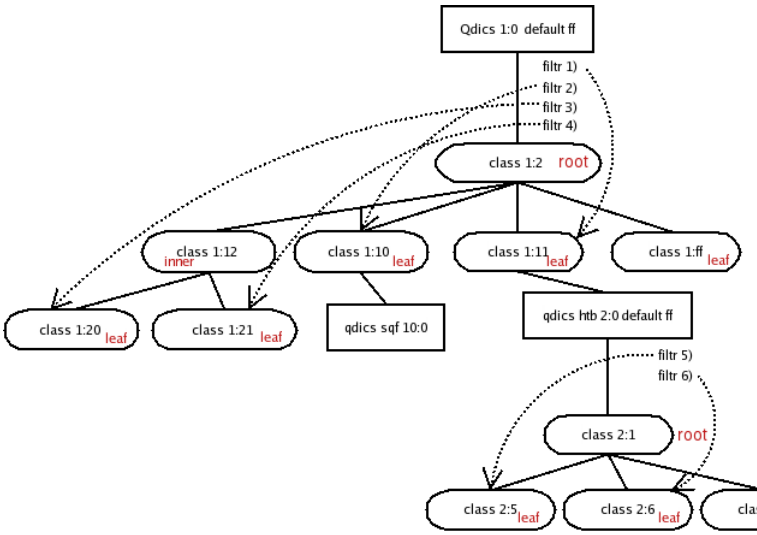
[7]

If it runs out of tokens, it waits until the supply of tokens is replenished and resumes network traffic while the tokens last. TBF is great if the administrator needs to control the *speed* of a network interface.

3.2 Classfull qdiscs

Classfull qdiscs are qdiscs that can contain multiple classes and each class can again contain multiple classes. They can be best imagined as *trees*. Each tree node can have a *filter*[1] assigned to it.

1. specifies how should a packet be processed, explained in chapter Filters



[21]

A more advanced classful qdisc example

They can be used if more *advanced shaping* is required. A tree with a root node and several other nodes and leafs can be created. Each node will have a minimum and maximum transmit rate assigned.

The table and its *leafs* by itself won't shape any traffic, but assigning a *filter* to each node will allow the administrator to specify, what kind of traffic should be limited by the minimum and maximum traffic rate of that leaf. If there are several computers in the network and bandwidth has to be guaranteed to a specific customer while not severely limiting others, shaping using classfull qdiscs is the ideal solution.

The 2.6.33 Linux kernel contains the *Class based queuing (CBQ)* and *Hierarchy token bucket (HTB)* classfull qdiscs. More information about each of these qdisc will be given in their own chapters.

3.2.1 CBQ

Class based queuing (CBQ) was the *first Linux network shaper*. It uses a simple idling mechanism, if the interface is to be throttled to 10% of its bandwidth, CBQ will simply disable the interface for 90% of the time.

As it was the first shaper, it served its purpose and brought advanced shaping to the Linux kernel that was not present in advanced routers of that time while proving Linux could get the job done.

But it has been outperformed by other newer shapers, such as *HTB*[1]. Even the author of CBQ noted that HTB performs better[2]. It still has some advantages in present times, as due to its simplified shaping algorithm it takes less CPU cycles than HTB. This however comes at the cost of accuracy.

The CBQ estimator uses the time between two sent packets to determine if the class is overloaded or not. This approach is inaccurate when used on wireless networks, as the time between two packets is highly variable and dependent on the connection.[11]

3.2.2 HTB

Hierarchy token bucket (HTB) is the successor of CBQ, as it outperforms it while providing *more accurate* shaping, the only disadvantage being a slightly higher CPU usage.

Martin Devera, the author of *HTB3.3*[2] has done extensive testing that can be read on his page about HTB[10].

HTB uses a more accurate estimator that allows precise shaping even on wireless networks or traffic that is being routed through a network tunnel. [11]

3.3 Traffic control with TC

Tc, short for *traffic control*, is used for manipulating traffic control settings such as *shaping* on Linux.

3.3.1 Getting information

Tc can be used to *get information* about *qdiscs*, *classes* and *filters* assigned to network interfaces. Each of these simple commands can print out the selected information about a network interface.

```
tc class show dev eth1
tc c s dev eth1
```

1. explained in chapter HTB
2. the currently used version of the algorithm being the third rewrite

```
tc filter show dev eth1
tc f s dev eth1

tc qdisc show
tc q s
```

The *first two pairs* of commands should *print out nothing* on a newly installed Linux distribution, as there are not any classes or filters assigned to any network interface. A device name *must be specified* for the commands to print out any output.

This is because simple networking does not need any classes or filters to function. Classes and filters are meant to add advanced networking capabilities that are primarily aimed at routers or servers.

The *third pair* of commands should print out information about *every* default qdisc that is assigned to each network interface. This is different from the first two pairs of commands, as a device name does not need to be specified when using the command.

```
qdisc pfifo_fast 0: dev eth0 root refcnt 2 \
 bands 3 priomap  1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1
```

This means that the device eth0 has the default qdisc pfifo_fast assigned to it, pfifo_fast has 3 bands and its priority map is set to 1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1.

The priomap maps all packets with a specific type of service to a specific pfifo_fast band. The priomap shown in the example is the default prio map that is assigned using the default priority table.

TOS	Means	Linux Priority	Band
0x0	Normal Service	0 Best Effort	1
0x2	Minimize Monetary Cost	1 Filler	2
0x4	Maximize Reliability	0 Best Effort	1
0x6	mmc+mr	0 Best Effort	1
0x8	Maximize Throughput	2 Bulk	2
0xa	mmc+mt	2 Bulk	2
0xc	mr+mt	2 Bulk	2

```
0xe     mmc+mr+mt               2 Bulk          2
0x10    Minimize Delay          6 Interactive   0
0x12    mmc+md                  6 Interactive   0
0x14    mr+md                   6 Interactive   0
0x16    mmc+mr+md               6 Interactive   0
0x18    mt+md                   4 Int. Bulk     1
0x1a    mmc+mt+md               4 Int. Bulk     1
0x1c    mr+mt+md                4 Int. Bulk     1
0x1e    mmc+mr+mt+md            4 Int. Bulk     1
```

A more expanded priority table along with more details about the priority qdisc can be read from the prio manual page by using the *man tc-prio* command.

Tc can also be used to shorten the scale of information to a specific network device, with a syntax similar that of which was used in the ip command examples.

```
tc qdisc show dev eth0
```

This command will show the qdisc information for the device eth0 only, this can be useful on larger and more complex systems.

3.3.2 Changing the default qdisc

Tc can be used to assign a new classless or classfull qdisc to an interface. The following example will assign a new classless qdisc to an existing interface. Examples of using classfull qdiscs will be given in chapter Managing classless qdiscs.

```
tc qdisc change dev eth2 root bfifo
```

This command will change the qdisc currently assigned to the eth2 interface to bfifo. Take note that this command will only function if a qdisc is assigned to the network interface, if an interface has *no qdisc(qdisc noop)* assigned, the command will return the *RTNETLINK answers: No such file or directory* error.

This can be the case of a newly created tunneling interface, which will not have a qdisc assigned until it is taken up for the first time, after which it will gain the *noqueue* qdisc.

3.3.3 Netem

Netem (network emulator) is a Linux kernel component that can emulate the properties of wide area networks (wan). It can add delay to incoming and outgoing packets, simulate packet loss, corruption, duplication, reordering and can be configured by the tc command.

3.3.4 Basic tc commands that are using Netem

All of the following examples can be also found on the *Linux Foundation* netem page[13].

The delay of a network interface can be increased with:

```
tc qdisc add dev eth0 root netem delay 100ms
```

This command will *delay* both incoming and outgoing packets on the interface by 100 milliseconds. The effect of this change is best noticed by connecting to to a remote server using SSH[1] and typing text in the terminal, the text typed will have a delay of at least 100 ms which is noticeable.

A random delay can also be assigned to an interface:

```
tc qdisc change dev eth0 root netem delay 100ms \
 10ms
```

This will enable the delay of 100 ms +- 10 ms, effectively setting the range of the delay from 90 to 110 ms on the interface eth0.

Tc can also be used to set the number of dropped packets on a network interface:

```
tc qdisc change dev eth0 root netem loss 0.1\%
```

1. a secure shell client that enables terminal access on a remote host

This will set a packet loss of 0.1% on the interface eth0. The interface will drop 1 out of 1000 packets.

A delay distribution can be be assigned to the qdisc.

Netem can take a table that specifies a non uniform distribution from the /usr/share/tc directory which contains the *normal*, *pareto* and *paretonormal* distributions by default.

```
tc qdisc change dev eth0 root netem delay 100ms \
 20ms distribution normal
```

3.3.5 Managing classless qdiscs

Tc can be used to change the qdisc that is assigned to a network interface.

```
tc qdisc del dev eth0 root
```

This command will delete the active qdisc on the interface eth0. Removing the active qdisc on a running interface will change the qdisc back to *pfifo_fast* (no change will be noticeable by the administrator). If the interface is down, *no qdisc will be assigned (noop)* until the interface is put to the up state. The *ip a dev eth0* command will return the following output.

```
2: eth0: <BROADCAST,MULTICAST> mtu 1500 qdisc \
 noop state DOWN qlen 1000
   link/ether 00:23:54:36:fd:f4 \
 brd ff:ff:ff:ff:ff:ff
```

Noop means there is no qdisc assigned to the network interface. If any interface with the noop qdisc is set to UP with *ip l set dev eth0 up*, the default *pfifo_fast* scheduler will take its place. This is the only way to add the default qdisc to a DOWN interface, as *tc qdisc replace dev eth0 pfifo_fast* will return the error message qdisc 'pfifo_fast' does not support option parsing.

3.4 Classes

Tc can create and assign classes to network interfaces. Several classes can be assigned to a single interface. By assigning classes, the administrator can create a complex class tree structure that can have filters assigned and can shape the network as the administrator sees fit.

For any of the class adding commands to work, the network interface must have a classless qdisc such as CBQ or HTB (HTB being the better choice due to performance and ease of use) assigned.

The

```
tc class
```

command can add, remove or view filters. An example of adding a new class to a network interface and viewing all the classes assigned to it will be given in chapter A simple shaping example.

3.5 Filters

Filters are *rules* that can be attached to each classfull qdisc node and specify how should a packet be processed based on specific criteria.

The most commonly used filter is *u32*, which is very powerful. It can filter packets based on several criteria, including *source* and *destination address*, *port* or *protocol*.

The

```
tc filter
```

command can *add*, *remove* or *view* filters. A simple example of adding a filter will be shown in chapter A simple shaping example. More advanced examples of use can be found in *The u32 filter*[1].

3.6 A simple shaping example

This chapter will show a simple example of assigning a classfull HTB qdisc to the default network interface (eth2 in this case) with a single leaf node and then assigning a filter leading all traffic to this node.

```
tc qdisc add dev eth2 root handle 1: htb
tc c add dev eth2 parent 1: classid 1:1 htb \
 rate 20kbit ceil 20kbit
tc f add dev eth2 protocol ip parent 1: prio 1 \
 u32 match ip dst 192.168.77.24 flowid 1:1
```

The *first command* will assign a root HTB qdisc to the *eth2* device.

The *second command* will assign a leaf node to the root HTB qdisc. This leaf has a limited transfer rate set (the ceil value is 20kbit) which severely limits all the network traffic diverted to this leaf by a filter.

The *third command* creates a filter that is applied to all ip packets that have the destination IP address 192.168.77.24. Each packet that matches the filter parameters will be diverted to the single leaf node 1:1. The end result is that all the ip network traffic heading to 192.168.77.24 will have a maximum transfer speed of 20kbit.

This configuration can simply be tested by creating a large file and transferring it to the 192.168.77.24 host via SCP.

```
time dd if=/dev/zero of=test.bin bs=100000000 \
 count=1 scp test.bin root@192.168.77.24:/home/
```

The first command will create a 100MB file that is full of zeroes. This file is ideal for network speed testing.

The second command will copy this file to the 192.168.77.24 hosts /home/ directory *(assuming the host is a Linux machine with an SSH server enabled and the firewall(iptables) properly configured)* via SCP[1]. The speed of the transfer is shown when the file is being copied and it should be 20kbit at maximum.

The *maximum transfer rate (ceil)* of the 1:1 class can be changed and the test can be repeated to see immediate results.

```
tc c change dev eth2 parent 1: classid 1:1 htb \
 rate 20kbit ceil 2000kbit
```

1. secure copy, an application for sending files via SSH

Tc can be used to create more complex network shapers that have large trees with each leaf having a different filter assigned, but those examples are beyond the scope of this guide and can be easily found on the Internet. One example is the *A Practical Guide to Linux Traffic Control*[6] guide.

The new changed class configuration can be viewed by using the following command.

```
tc class show dev eth2
```

This will print out the information about the newly added shaper classes.

```
class htb 1:1 root prio 0 rate 20000bit \
 ceil 2000Kbit burst 1600b cburst 1600b
```

A similar command can be used to view the filters. The command

```
tc filter show dev eth1
```

will print out a list of filters that are assigned to the eth1 interface.

```
filter parent 1: protocol ip pref 1 u32
filter parent 1: protocol ip pref 1 u32 fh \
 800: ht divisor 1
filter parent 1: protocol ip pref 1 u32 fh \
 800::800 order 2048 key ht 800 bkt 0 flowid 1:1
  match c0a84d18/ffffffff at 16
```

This shows the current filter configuration of the simple shaper that was created at the beginning of this chapter.

These are just basic examples of using the tc command and more information can be gained by either reading the tc manual page *(man tc)* or looking up more specialized guides that are aimed only on tc.

4 Other iproute2 tools

4.1 Rtmon

Rtmon is a routing monitor that listens on netlink sockets and monitors changes in the routing tables. It can be used to create a log of all the routing table changes.

```
rtmon file rtlog.log
```

This command will make rtmon log all the routing table changes and print them in the rtlog.log file. The log file generated by rtmon is only readable by using ip monitor. The command

```
ip monitor rtlog.log
```

will print out the contents of the log file.

4.2 Socket statistics

Socket statistics (SS) is a utility that can investigate sockets and print out information about sockets that match selected criteria.

```
ss -s
```

will print out the statistic summary of the currently existing sockets.

```
ss -l
```

will show the list of currently open sockets.

```
ss -pl
```

will show the processes that are using the sockets.

```
ss -o state established '( dport = :http or \
```

```
sport = :http )'
```

will show all the existing http connections. *Dport* can be *http*, *ssh*, *scp* or several other types of connection.

```
ss -o state close-wait
```

will print out the list of sockets in the close-wait state. Any socket state can be used.

state can be *established*, *Syn-sent*, *Syn-Recv*, *Fin-wait1*, *Fin-wait2*, *Time-wait*, *Close-wait* and *Closed*.

Multiple states can be defined:

```
ss -o state close-wait state established
```

The following command will dump out the raw information to the file named test.

```
ss -D test
```

5 Summary

This work was aimed at gathering information about the use of the iproute2 package and promoting its use through a limited number of working commands. These commands can be either used by a novice administrator or an experienced user to successfully manage the networking aspect of Linux or simply test their functionality by creating simple model examples such as simple tunnels or a shaper with a single class.

The list of commands present in this manual is not absolute and there are many more commands to explore, but explaining or at least mentioning every single option is not feasible as the length of this work could intimidate a new reader. The second reason is that most of the command options are present in the manual and the missing options will probably be added after this guide is published. Therefore it is left for the reader to look up additional uses if the manual sparked an interest in this topic or iproute2 itself.

The information present in this manual can be used to improve and expand existing iproute2 documentation and manual pages. One such example is the ip ntable command, as the ip command manual pages do not contain any reference to this iproute2 capability. One of the benefits of this work is therefore the possible addition of usage examples and output information about this command to the manual pages of ip.

This manual was written natively in *Latex*[1], therefore it can be easily converted to HTML or other formats depending on the readers preference and can be easily published on the Internet.

The work was written in English so that it could reach a wider target audience and could benefit the whole Linux community not just our local Czech and Slovak colleagues.

This work could be expanded in the future to contain additional examples of use depending on the wishes of readers. Specific chapters could be expanded if new iproute2 functionality is added.

---

1. a document markup language and document preparation system

Bibliography

[1] b42.cz. The u32 filter. [online] http://b42.cz/notes/u32_classifier/, December 2010.

[2] Leonardo Balliache. Htb queuing discipline. [online] http://www.opalsoft.net/qos/DS-28.htm, December 2010.

[3] Klaus Wehrle; Frank Pählke; Hartmut Ritter; Daniel Müller; Marc Bechler. The linux® networking architecture: Design and implementation of network protocols in the linux kernel. [online] http://www.6test.edu.cn/~lujx/linux_networking/0131777203_ch15lev1sec3.html, December 2010.

[4] Christian Benvenuti. Understanding linux network internals. [online] http://books.google.com/books?id=w3EHS2cobBAC&pg=PT791&lpg=PT791&dq=app_probes&source=bl&ots=baqLaSyjfm&sig=oI9aDgvVOprM6VmjEobetSUgWn8&hl=sk&ei=Jva9TPXiNsq6jAeAybGIAg&sa=X&oi=book_result&ct=result&resnum=3&ved=0CCEQ6AEwAg#v=onepage&q=gc_int&f=false, December 2010.

[5] K. Nichols; S. Blake; F. Baker; D. Black. Definition of the differentiated services field (ds field) in the ipv4 and ipv6 headers. [online] http://www.ietf.org/rfc/rfc2474.txt, December 2010.

[6] Jason Boxman. A practical guide to linux traffic control. [online] http://blog.edseek.com/~jasonb/articles/traffic_shaping/classes.html, December 2010.

[7] Martin A. Brown. Classless queuing disciplines (qdiscs). [online] http://linux-ip.net/articles/Traffic-Control-HOWTO/classless-qdiscs.html, December 2010.

[8] Martin A. Brown. Guide to ip layer network administration with linux. [online] http://linux-ip.net/html/routing-rpdb.html, December 2010.

[9] Martin A. Brown. Ip command reference. [online] http://linux-ip.net/gl/ip-cref/node16.html, December 2010.

[10] Martin Devera. Htb 3 performance compared. [online] http://luxik.cdi.cz/~devik/qos/htb/htb3perf/cbqhtb3perf.htm, December 2010.

[11] Martin Devera. Princip a uziti htb qos discipliny. [online] http://luxik.cdi.cz/~devik/qos/htb/cs/slt02htb.ps, December 2010. [in Czech language].

[12] R. Draves. Default address selection for internet protocol version 6 (ipv6). [online] http://www.ietf.org/rfc/rfc3484, December 2010.

[13] Linux Foundation. netem. [online] http://www.linuxfoundation.org/collaborate/workgroups/networking/netem, December 2010.

[14] The Linux Foundation. net-tools. [online] http://www.linuxfoundation.org/collaborate/workgroups/networking/net-tools, December 2010.

[15] Thomas Graf. Extended matches api. [online] http://lxr.free-electrons.com/source/net/sched/ematch.c?v=2.6.25;a=arm, December 2010.

[16] IBM. Tcp maximum segment size tuning. [online] http://publib.boulder.ibm.com/infocenter/pseries/v5r3/index.jsp?topic=/com.ibm.aix.prftungd/doc/prftungd/tcp_max_seg_size_tuning.htm, December 2010.

[17] Samuel J. Leffler; Michael J. Karels. Trailer encapsulations. [online] http://www.ietf.org/rfc/rfc893, December 2010.

[18] S. Kent. Ip authentication header. [online] http://www.ietf.org/rfc/rfc4302, December 2010.

[19] S. Kent. Ip encapsulating security payload (esp). [online] http://www.ietf.org/rfc/rfc4303, December 2010.

[20] Nathan Lutchansky. Linux 6rd howto. [online] http://www.litech.org/6rd/, December 2010.

[21] mrak.cz. Htb principy a priklady. [online] http://mrak.cz/page/veci/htb_princip_detaily_priklad.php, December 2010. [in Czech language].

[22] David C. Plummer. Rfc 826. [online] http://tools.ietf.org/html/rfc826, December 2010.

[23] Pavel Satrapa. 6rd - nový koncept nasazení ipv6. [online] http://www.lupa.cz/clanky/6rdnbspndash-novy-koncept-nasazeni-ipv6/, December 2010. [in Czech language].

[24] Pekka Savola. Ipv6 multicast deployment issues. [online] http://www.6net.org/publications/standards/draft-savola-v6ops-multicast-issues-01.txt, December 2010.

[25] S. Kent; K. Seo. Security architecture for the internet protocol. [online] http://www.ietf.org/rfc/rfc4301, December 2010.

[26] Charles Spurgeon. What is sqe test, and when to use it? [online] http://www.ethermanage.com/ethernet/sqe/sqe.html, December 2010.

[27] Cisco Systems. Rx fifo sync errors. [online] http://www.cisco.com/en/US/products/hw/switches/ps525/products_tech_note09186a0080094170.shtml, December 2010.

[28] D. Farinacci; T. Li; S. Hanks; D. Meyer; P. Traina. Generic routing encapsulation (gre). [online] http://www.ietf.org/rfc/rfc2784, December 2010.