IPROUTE2 Utility Suite Howto

Main PolicyRouting.Org Website PolicyRouting Book

9.0 Obtaining & Compiling IPROUTE2

9.1 IP Command Set

9.2 ip address - protocol address management

9.3 IP Interface Primary and Secondary Addressing:

9.4 ip neighbour --- neighbour/arp table management.

9.5 ip route - routing table management.

IPROUTE2 Utility Suite Documentation

This docvumentation covers the ip utility from IPROUTE2. This utility is

written by Alexey N. Kuznetsov who also wrote the IPv6 and IPv4 routing code for Linux 2.2. This is the utility he uses for manipulating the Linux 2.2-2.6 network interface code.

We will begin by explaining where to obtain the utility collection and how to compile it. After it is compiled we will cover the utilities created and in what location on the system they should reside. This includes all of the utilities in the IPROUTE2 suite.

Then we will begin extensive coverage of the ip command with documentation of usage and examples. This section draws heavily upon Alexey's own documentation of the command with additional discussion and examples. Some of the usages of the command, such as multicast and IPv6 specific usage will be deferred at this point but we will be extending this document with that coverage as time goes on. While this is often what would be found in man pages, no man pages currently exist for the ip command and Alexey's own current documentation is only available in Latex format. With Alexey's permission we have edited and expanded the Latex documentation into the sections found here. If there are errors in these sections they probably belong to Matthew's translation and should be addressed to him first.

To tie together what we have learned about the ip utility we will list a few working examples of the ip utility. These include several longer script examples from Alexey along with some daily usage features of the utility. We then in the Table of Contents list a set of examples from real life that are collected here.

Obtaining & Compiling IPROUTE2

The ip utility is just one of the utilities in the IPROUTE2 utility package from Alexey. The primary FTP site was located in Russia at ftp://ftp.inr.ac.ru/ip-routing/ but is no longer running. The most complete mirror is located at http://www.linuxgrill.com/anonymous/iproute2/ with the newest OSDL source code located within the http://www.linuxgrill.com/anonymous/iproute2/NEW-OSDL/ directory. We

will assume that you have obtained the latest package usually called iproute2-current symlinked to the latest dated version. The version we primarily cover here is the 1999-06-30 version of IPROUTE2.

Once the utility has been obtained you need to unpack it into whatever directory you use for compiling source code. The default is to use /usr/src. When you have the package untarred you can enter the directory and just type make. You must have the kernel source code that was used to compile your current running kernel located in /usr/src/linux. You do want to compile a version of your own unless you are using a distribution that includes the utility and you have not remade your kernel. Since one of the best tuning and security functions you can perform on your system is to obtain and compile your own specific kernel you will want to compile this utility also as it is the single most important utility in the IP configuration of your system.

After you have typed make the utility suite will compile. Then we have to install the various parts. There is no install target in the makefile. All of the utilities in this package should be installed into the /sbin directory. This is so that they are available even before your /usr directory is mounted. There is additionally a /etc/iproute2 directory in the package that contains sample definition files. If you do not have a /etc/iproute2/ directory on your system then create one and copy the contents of the package directory to the new directory. If an /etc/iproute2/ directory exists and you do not know what it is being used for then you will want to find out if the files in that directory have some meaning to the system you are running. If not then replacing them with the files in the package directory will not hurt.

In a nutshell we want to perform the following steps:

1. Compile the utilities by typing make

2. Check /etc/iproute2/ with ls -l /etc/iproute2

3. If needed create /etc/iproute2/ with

mkdir /etc/iproute2/

4. Populate it with cp ./etc/iproute2/* /etc/iproute2/

5. Change into the ip directory with cd ip

6. cp ifcfg ip routef routel rtacct rtmon rtpr /sbin

7. Change into tc directory with cd ../tc

8. cp tc /sbin

This will compile the utility and copy the configuration files and the executables into the appropriate directories. We should now be able to execute the ip utility from anywhere on the system by typing ip. To test and see if this worked type ip addr and you should get a list of the interfaces and addresses on your system.

IP Command Set

In this section we will present a comprehensive description of the ip utility from Alexey Kuznetsov's IPROUTE2 package. We will start by going through most of the ip command in extreme detail. We will cover the link, addr, route, rule, neigh, tunnel, and monitor parts of the ip command. The multicast sections will be covered in a "to be added later" section on IPv6 and multicasting.

We will first go through all of the command syntax of the ip command. This is due to the situation, current as of February 2000, that there are no man pages for ip and the documentation is only available in Latex format. If you have read the ip-cref.tex document that Alexey has written as included in 1999-06-30 distribution of IPROUTE2 then feel free to just skim through most of this section. Matthew has extended the discussion and examples somewhat but the core is taken from ip-cref.tex. If you have any questions or comments about the examples or statements in this section please direct them to Matthew. Note also that by the time you read this the ip command may have changed for 2.3/2.4. As it changes we will attempt to keep this document current.

IP Global Command Syntax

The generic form of the ip command is

ip [OPTIONS] OBJECT [COMMAND [ARGUMENTS]]

OPTIONS:

OPTIONS is a multivalued set of modifiers that affect the general behaviour and output of the ip utility. All options begin with the "-" character and may be used both in long and abbreviated form. Currently the following options are available

-V, -Version --- print the version of the ip utility and exit.

-s, -stats, -statistics --- output more information.

This option may be repeated to increase the verbosity level of the output. As a rule the additional information is device or function statistics or values. In many cases the values output should be considered in the same sense as output from the /proc/ directory where the name of the value is not directly related to the value itself. See later when we run this option with different network device drivers.

-f, -family {inet, inet6, link} --- enforce which protocol family to use.

If this option is not present, the protocol family output to use is guessed from the other command line arguments. If the rest of command line does not provide sufficient information to guess a protocol family, the ip command falls back to a default family of inet in the case of network protocols or to any. Link is a special family identifier meaning that no networking protocol is involved. There are several shortcuts for this option and they are as listed here:

-4 --- shortcut for -family inet.

-6 --- shortcut for -family inet6.

-0 --- shortcut for -family link.

-o, -oneline --- format the output records as single lines by replacing any line feeds with the "\" character.

This option is to provide a convenient method for sending the command output through a pipe. IE: When you want to count the number of output records with wc or you want to to grep through the output. As of 1999-06-30 the IPROUTE2 utility package includes the trivial script rtpr to convert the output back to the original readable form.

-r, -resolve --- use system name resolution to output DNS names

Do not use this option if you are reporting bugs with the ip utility or querying for usage advice. ip itself never uses DNS to resolve names to addresses. This option exists for the administrators convenience only.

OBJECT:

OBJECT is the object type on which you wish to operate on or obtain information about. The object types understood by the current ip utility are link, address, neighbor, route, rule, maddress, mroute, and tunnel.

link --- physical or logical network device.

address --- protocol (IPv4 or IPv6) address on a device.

neighbour --- ARP or NDISC cache entry.

route --- routing table entry.

rule --- rule in routing policy database.

maddress --- multicast address.

mroute --- multicast routing cache entry.

tunnel --- tunnel over IP.

The names of all of the objects may be written in full or abbreviated form. IE: address may be abbreviated as addr or just a. However if you use these commands within scripts you should make it a habit to always use the full specification of the action. Using the abbreviation makes it easy to use on the command line but hard to understand the logic within scripts. Since you may not be the only person who ever has to deal with your scripts then you should strive to make them as complete as possible.

COMMAND:

COMMAND specifies the action to perform on the object. The set of possible actions depends on the object type. Typically it is possible to add, delete, and show (list) the object(s), but some objects will not allow all of these operations and many have additional actions and commands. Note that the command syntax help which is available for all objects prints out the full list of available commands and argument syntax conventions. If no command is given a default command is assumed. The default command is usually show (list) but if the objects of the class cannot be listed then the default is to print out the command syntax help.

ARGUMENTS:

ARGUMENTS is the list of command options specific to the command. The arguments depend on the command and the object. There are two types of arguments that can be issued:

--- flags - which are abbreviated with a single keyword

--- parameters - consisting of a keyword followed by a value

Each command has a default parameter which is used if the arguments are omitted. IE: The dev parameter is the default for the ip link command thus ip link list eth0 is equivalent to ip link list dev eth0. Within all the command descriptions below we distinguish default parameters with the marker (default).

As we mentioned above for the names of objects, all keywords may be

abbreviated with the first or first few unique letters. These shortcuts are convenient when ip is used interactively, but they are not recommended for use in scripts and please do not use them when reporting bugs or asking for help. Officially allowed abbreviations are listed along with the first mention of the command.

Error Conditions

The ip command most commonly fails for the following reasons:

* Wrong command line syntax

This is often due to using an unknown keyword, a wrongly formatted IP address, wrong keyword argument for the command, etc. In this case the ip command exits without performing any actions and prints out an error message containing information about the reason for failure. In some cases it prints out the command syntax help.

* The arguments did not pass self-consistency verification

* ip failed to compile a kernel request from the arguments due to insufficient user provided information

* Kernel returned an error to a syscall. In this case ip prints the error message as it was output from perror(3), prefixed with a comment and the syscall identifier.

* Kernel returned an error to a RTNETLINK request. In this case ip prints the error message as it was output from perror(3) prefixed with "RTNETLINK answers".

Note that all ip command operations are atomic. This means that if the ip command fails it does not change anything in the system. One harmful exception is the ip link command which may change only part of the device parameters given on the command line. We will mention this again in the section on ip link usage and reccomend that all ip link actions be performed individually. This is actually a preferred use for the ip command

in general. If you need to perform many repetitions of the command use a script loop or a script as then any generated error messages can be associated with the appropriate ip command action.

It is difficult to list all possible error messages especially the syntax errors. As a rule their meaning should be clear from the context of the command that was issued. For example if we issue the command ip link sub eth0 with the obvious misspelling of set then we get the error message "Command "sub" is unknown, try "ip link help"" which should prompt us to check our command syntax.

In using the ip command there are several facilities that need to be present in order for the command to perform its functions. The ip command talks to the kernel through the NETLINK interface. This is turned on by the NETLINK options which are enabled in the kernel compile. If the ip command does not work or you get an error message then you may not have the needed functions defined or your kernel is not the one you compiled. The most common mistakes are:

* NETLINK is not configured in the kernel. The error message is

"Cannot open netlink socket Invalid value"

* RTNETLINK is not configured in the kernel.

In this case one of the following messages may be printed depending on the actual command issued:

"Cannot talk to rtnetlink Connection refused"

"Cannot send dump request Connection refused"

ip link - network device configuration

A link refers a network device. The ip link object and the corresponding command set allows viewing and manipulating the state of network devices. The commands for the link object are just two, set and show.

ip link set --- change device attributes.

Abbreviations: set, s

Warning

You can request multiple parameter changes with ip link. If you request multiple parameter changes and any ONE change fails then ip aborts immediately after the failure thus the parameter changes previous to the failure have completed and are not backed out on abort. This is the only case where using the ip command can leave your system in an unpredictable state. The solution is to avoid changing multiple parameters with one ip link set call. Use as many individual ip link set commands as necessary to perform the actions you desire.

Arguments:

* dev NAME (default) --- NAME specifies the network device to operate on

* up / down --- change the state of the device to UP or to DOWN

* arp on / arp off --- change NOARP flag status on the device

Note that this operation is not allowed if the device is already in the UP state. Since neither the ip utility nor the kernel check for this condition, you can get very unpredictable results changing the flag while the device is running. It is better to set the device down then issue this command.

* multicast on / multicast off --- change MULTICAST flag on the device.

* dynamic on / dynamic off --- change DYNAMIC flag on the device.

* name NAME --- change name of the device.

Note that this operation is not recommended if the device is running or has some addresses already configured. You can break your systems security and screw up other networking daemons and programs by

changing the device name while the device is running or has addressing assigned.

* txqueuelen NUMBER / txqlen NUMBER --- change transmit queue length of the device

* mtu NUMBER --- change MTU of the device.

* address LLADDRESS --- change station address of the interface.

* broadcast LLADDRESS, brd LLADDRESS or peer LLADDRESS --- change link layer broadcast address or peer address in the case of a POINTOPOINT interface

Note that for most physical network devices (Ethernet, TokenRing, etc) changing the link layer broadcast address will break networking. Do not use this argument if you do not understand what this operation really does.

* The ip command does not allow changing the PROMISC or ALLMULTI flags as these flags are considered obsolete and should not be changed administratively.

Examples:

ip link set dummy address 000000000001 --- change station address of the interface dummy.

ip link set dummy up --- start the interface dummy.

ip link show --- look at device attributes.

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

Arguments:

* dev NAME (default) --- NAME specifies network device to show.

If this argument is omitted, the command lists all the devices.

* up --- display only running interfaces.

Output:

kuznet@alisa~:$ ip link ls dummy

2: dummy: <BROADCAST,NOARP> mtu 1500 qdisc noop

link/ether 000000000000 brd ffffffffffff

The number followed by a colon is the interface index or ifindex. This number uniquely identifies the interface. If you look at the output from cat /proc/net/dev you will see that the network devices are listed in the same order as the numbering you see here. After the ifindex is the interface name (eth0, sit0 etc.). The interface name is also unique at any given moment, however interfaces may disappear from the list, such as when the corresponding driver module is unloaded, and another interface with the same name will be created later. Additionally with the ip link set DEVICE name NEWNAME command the system administrator may change the devices name.

The interface name may also have another name or the keyword NONE appended after an "@" sign. This signifies that this device is bound to another device in a master/slave device relationship. Thus packets sent through this device are encapsulated and forwarded on via the master device. If the name is NONE, then the master device is unknown.

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

The qdisc (queuing discipline) shows which queuing algorithm is used on the interface. In particular the keyword noqueue means that this interface does not queue anything and the keyword noop indicates that the interface is in blackhole mode in which all of the packets sent to it are immediately discarded.

The qlen indicates the default transmit queue length of the device measured in packets.

Following all of this inormation is a section within angle brackets. Within the angle brackets is where the interface flags are summarized. The most applicable flags are as follows:

UP --- this device is turned on, ready to accept packets for transmission onto the network and it may receive packets from other nodes on the network.

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

BROADCAST --- this device has the facility to send packets to all other hosts sharing the same physical link. Example: Ethernet

POINTOPOINT --- the network has only two ends with two nodes attached. All the packets sent to the link will reach the peer link and all packets received are origined by the peer.

If neither LOOPBACK nor BROADCAST nor POINTOPOINT are set, the interface is assumed to be a NBMA (Non-Broadcast Multi-Access) link. NBMA is the most generic type of device and also the most complicated type of device because a host attached to a NBMA link cannot send information to any other host without additional manually provided configuration information.

MULTICAST --- an advisory flag noting the interface is aware of multicasting. Broadcasting is particular case of multicasting where the multicast group contains all of the nodes on the link as members. Note that software must NOT interpret the absence of this flag as the incapability of the interface to multicast. Any POINTOPOINT and BROADCAST link is multicasting by definition because we have direct access to all the link neighbours and thus to any particular group of them. The use of high bandwidth multicast transfers is not recommended on

broadcast-only networks due to the high expenses associated with the transmission, but such use is not strictly prohibited.

PROMISC --- the device listens and feeds to the kernel all of the traffic on the link. This includes every packet on the network that passes our transceiver. Usually this mode exists only on broadcast links and is used by bridges and network monitoring devices.

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

NOARP --- this flag is different from the other flags. It has no invariant value and its interpretation depends on network protocols involved. As a rule it indicates that the device does not need any address resolution and that the software or hardware knows how to deliver packets without any help from the protocol stacks.

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

SLAVE --- this interface is bonded to other interfaces in order to share link capacities.

Other flags do exist and can be seen in within the angle brackets but they are either obsolete (NOTRAILERS), not implemented (DEBUG), or specific to certain devices (MASTER, AUTOMEDIA and PORTSEL). We will not discuss them here. Additionally the values of the PROMISC and ALLMULTI flags as shown by the ifconfig utility and by the ip utility are different. The ip link list command provides the current true device state, whereas ifconfig shows the flag state which was set through ifconfig itself.

The second line of the output from the example contains information about the link layer addresses associated with the device. The first word (ether, sit) defines the interface hardware type which then determines the format and semantics of the addresses and thus logically is part of the address itself. The default format of station and broadcast addresses (or peer addresses for pointopoint links) is a sequence of hexadecimal bytes

separated by colons. However some link types may instead have their own natural address formats which are used in the presentation. IE: The addresses of IP tunnels are printed as dotted-quad IP addresses. While NBMA links have no well-defined broadcast or peer address, this field may contain useful information such as the address of a broadcast relay or the address of an ARP server. Multicast addresses are not shown by this command, see ip maddr list output.

When given the option -statistics ip will print the interface statistics as additional information in the listing. Note that you can give this option multiple times with each repetition increasing the verbosity of output.

kuznet@alisa~ $ ip -s link ls eth0

3: eth0: <BROADCAST,MULTICAST,UP> mtu 1500 qdisc cbq qlen 100

link/ether 00a0cc661878 brd ffffffffffff

RX bytes packets errors dropped overrun mcast

2449949362 2786187 0 0 0 0

TX bytes packets errors dropped carrier collsns

178558497 1783945 332 0 332 35172

The RX and TX lines summarize receiver and transmitter statistics. The information output breaks down into:

bytes --- total number of bytes received or transmitted on the interface.

This number wraps when the maximal length of the natural data type on the architecture is exceeded. In order to provide correct long term data from this output these statistics should be continuously monitored. Continuous monitoring of this data requires a user level daemon to sample the output periodically.

packets --- total number of packets received or transmitted on the

interface.

errors --- total number of receiver or transmitter errors.

dropped --- total number of packets dropped because of lack of resources.

overrun --- total number of receiver overruns resulting in packet drops. As a rule if the interface is overrun you have a serious problem either within the kernel or your machine is too slow to handle the speed of this interface.

mcast --- total number of received multicast packets. This option is supported only on certain devices.

carrier --- total number of link media failures such as those due to lost carrier.

collsns --- total number of collision events on Ethernet-like media. This number has different interpretations on other link types.

compressed --- total number of compressed packets. It is available only for links using VJ header compression.

When you issue the -statistics option more than once you get additional output depending on the statistics supported by the device itself as in the following example with Ethernet:

kuznet@alisa~ $ ip -s -s link ls eth0

3: eth0: <BROADCAST,MULTICAST,UP> mtu 1500 qdisc cbq qlen 100

link/ether 00a0cc661878 brd ffffffffffff

RX bytes packets errors dropped overrun mcast

2449949362 2786187 0 0 0 0

RX errors length crc frame fifo missed

0 0 0 0 0

TX bytes packets errors dropped carrier collsns

178558497 1783945 332 0 332 35172

TX errors aborted fifo window heartbeat

0 0 0 332

In this case the error names are pure Ethernetisms. Other devices may have non-zero fields in these positions but the headers are generated independantly of the device responses. It is up to the device driver to send more appropriate error messages to the system logging facility such as is done by the TokenRing driver.

ip address - protocol address management

Abbreviations: address, addr, a

Arguments: add, delete, flush, show (list)

The address refers to a protocol (IP or IPv6) address attached to a network device. Each device must have at least one address in order to use the corresponding protocol. It is possible to have several different addresses attached to one device. These addresses are not discriminated within the protocol structure so that the term alias is not quite appropriate for such multiple addresses and we will not refer to this situation in those terms.

The ip addr command allows you to look at the addresses and their properties on an interface. You can add new addresses and delete old ones without regard to any ordering. Later on we will discuss the concept of primary and secondary addresses as applied to Linux.

ip address add --- add new protocol address.

Abbreviations: add, a

Arguments:

dev NAME --- name of the device to which we add the address

local ADDRESS (default) --- address of the interface.

The format of the address depends on the protocol. IPv4 uses dotted quad and IPv6 uses a sequence of hexadecimal halfwords separated by colons. The ADDRESS may be followed by a slash and a decimal number, which encodes network prefix (netmask) length in CIDR notation. If no CIDR netmask notation is specified then the command assumes a host (/32 mask) address is specified.

peer ADDRESS--- address of remote endpoint for pointopoint interfaces. Again, the ADDRESS may be followed by a slash and decimal number, encoding the network prefix length. If a peer address is specified then the local address cannot have a network prefix length as the network prefix is associated with the peer rather than with the local address. In other words, netmasks can only be assigned to peer addresses when specifying both peer and local addresses.

broadcast ADDRESS --- broadcast address on the interface.

The special symbols "+" and "-" can be used instead of specifying the broadcast address. In this case the broadcast address is derived by either setting all of the interface host bits to one (+) or by setting all of the interface host bits to zero (-). In most modern implementations of IPv4 networking you will want to use the (+) setting. See the ipup init script in Chapter 15. Unlike ifconfig, the ip command does not set a broadcast address unless explicitly requested.

label NAME --- Each address may be tagged with a label string.

In order to preserve compatibility with Linux-2.0 net aliases, this string must coincide with the name of the device or must be prefixed with device name followed by a colon. (eth0:duh)

scope SCOPE_VALUE --- scope of the area within which this address is valid.

The available scopes are listed in the file

/etc/iproute2/rt_scopes. The predefined scope values are:

global --- the address is globally valid.

site --- (IPv6 only) address is site local, valid only inside this site.

link --- the address is link local, valid only on this device.

host --- the address is valid only inside this host.

Examples:

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

--- adds the usual loopback address to loopback device. The device must be enabled before this address will show up.

ip addr add 10.0.0.1/24 brd + dev eth0

--- adds address 10.0.0.1 with prefix length 24 (netmask 255.255.255.0) and standard broadcast to interface eth0

ip address delete --- delete protocol address.

Abbreviations: delete, del, d

Arguments:

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

Examples:

ip addr del 127.0.0.1/8 dev lo

--- deletes the loopback address from loopback device.

Alexey states:

"It would be better not to try to repeat this experiment 8-}"

Delete all IPv4 addresses on interface eth0:

while ip -f inet addr del dev eth0; do

nothing

done

Another method to disable IP on an interface using ip addr flush is discussed later.

ip address show --- look at protocol addresses.

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

Arguments:

dev NAME (default) --- name of the device.

scope SCOPE_VAL --- list only addresses with this scope.

to PREFIX --- list only addresses matching this prefix.

label PATTERN --- list only addresses with labels matching the PATTERN.

PATTERN is the usual shell regexp style pattern.

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

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

deprecated --- (IPv6 only) list only deprecated addresses.

primary / secondary --- list only primary (or secondary) addresses.

Example:

kuznet@alisa~ $ ip addr ls eth0

3: eth0: <BROADCAST,MULTICAST,UP> mtu 1500 qdisc cbq qlen 100

link/ether 00a0cc661878 brd ffffffffffff

inet 193.233.7.90/24 brd 193.233.7.255 scope global eth0

inet6 3ffe2400012a0ccfffe661878/64 scope global dynamic

valid_lft forever preferred_lft 604746sec

inet6 fe802a0ccfffe661878/10 scope link

The first two lines coincide with the output of ip link list as it is only natural to interpret link layer addresses as being addresses of the protocol family AF_PACKET. The list of IPv4 and IPv6 addresses follows accompanied by additional attributes such as scope value, flags, and address label. Address flags are set by the kernel and cannot be changed administratively. Currently the following flags are defined:

secondary --- this address is not used when selecting the default source address for outgoing packets. An IP address becomes secondary if another address within the same prefix (network) already exists. The first address within the prefix is primary and is the tag address for the group of all the secondary addresses. When the primary address is deleted all of the secondaries are purged too. See the examples for the actual functionality of these steps.

dynamic --- the address was created due to stateless autoconfiguration. In this case the output also contains information on the times for which the address remains valid. After the preferred lifetime (preferred_lft)

expires the address is moved to the deprecated state and after the valid lifetime (valid_lft) expires the address is finally invalidated.

deprecated --- the address is deprecated. It is still valid but cannot be used by newly created connections. See dynamic above.

tentative --- the address is not used because duplicate address detection is still not complete or has failed.

IP Interface Primary and Secondary Addressing:

To explain the actual relationship between primary and secondary addresses we will run the following experiment.

ip addr add 10.1.1.1/24 dev dummy

ip addr add 10.1.1.2/24 dev dummy

Now look at the output:

ip addr list dummy

3: dummy: <BROADCAST,MULTICAST,NOARP> mtu 1500 qdisc noop

link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff

inet 10.1.1.1/24 scope global dummy

inet 10.1.1.2/24 scope global secondary dummy

Now add in some addresses still in that network but add them as host addresses:

ip addr add 10.1.1.3/32 dev dummy

ip addr add 10.1.1.4/25 dev dummy

And run our list command:

ip addr list dummy

3: dummy: <BROADCAST,MULTICAST,NOARP> mtu 1500 qdisc noop

link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff

inet 10.1.1.1/24 scope global dummy

inet 10.1.1.3/32 scope global dummy

inet 10.1.1.4/25 scope global dummy

inet 10.1.1.2/24 scope global secondary dummy

And finally delete the primary address

ip addr del 10.1.1.1/24 dev dummy

Run the list command:

ip addr list dummmy

3: dummy: <BROADCAST,MULTICAST,NOARP> mtu 1500 qdisc noop

link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff

inet 10.1.1.3/32 scope global dummy

inet 10.1.1.4/25 scope global dummy

Note that the most important part of what we said above about secondary and primary addresses is the prefix (netmask) length. Even though technically you can consider the address 10.1.1.3 to belong within the network prefix 10.1.1.0/24, the actual prefix associated with the address is /32 so this address is treated independantly of the initial primary address. If you are still uncertain about why sit down and calculate out the networks and masks of the example above.

What we are showing here is that unlike the behaviour in the 2.0 series kernels under the horrid eth0:xx style aliasing is that multiple addresses on an interface are not neccesarily related. So if you want to (and we will

show an example in the howto section) you can enter in all of your ip addresses without network masks and treat them completely independantly.

ip address flush --- flush protocol addresses.

Abbreviations: flush, f

Arguments:

This commands flushes protocol addresses selected by some criteria. This command has the same arguments as show. The major difference is that this command will not run if no arguments are given. Otherwise you could delete all of your addresses by mistake. This command (and the other flush commands described below) are very dangerous. If you make a mistake the command does not ask or forgive but really will creully purge all of your addresses. Be warned!

With the option -statistics the command becomes verbose and prints out the number of deleted addresses and number of processing rounds made in order to flush the address list. If the -statistics option is given twice then ip addr flush also dumps all of the deleted addresses in the full format as described in the ip addr list section.

Examples:

Delete all the addresses from private network 10.0.0.0/8:

netadm@amber~ # ip -stat -stat addr flush to 10/8

2 dummy inet 10.7.7.7/16 brd 10.7.255.255 scope global dummy

3 eth0 inet 10.10.7.7/16 brd 10.10.255.255 scope global eth0

4 eth1 inet 10.8.7.7/16 brd 10.8.255.255 scope global eth1

*****Round 1, deleting 3 addresses*****

Flush is complete after 1 round

Another instructive example is deleting all IPv4 addresses from all Ethernet interfaces in the system:

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

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

netadm@amber~ # ip -6 addr flush dynamic

ip neighbour --- neighbour/arp table management.

Abbreviations: neighbour, neighbor, neigh, n

The neighbour table objects establish bindings between protocol addresses and link layer addresses for hosts sharing the same physical link. Neighbour object entries are organized into tables. The IPv4 neighbour object table is known under another name as the ARP table. These commands allow you to look at the neighbour table bindings and their properties, to add new neighbour table entries, and to delete old ones.

Arguments:

add, change, replace, delete, flush and show (list)

ip neighbour add --- add new neighbour entry

ip neighbour change --- change existing entry

ip neighbour replace --- add new or change existing entry

add, a; change, chg; replace, repl

These commands create new neighbour records or update existing ones.

to ADDRESS (default) --- protocol address of the neighbour. It is either an IPv4 or IPv6 address.

dev NAME --- the interface to which this neighbour is attached

lladdr LLADDRESS --- link layer address of the neighbour. LLADDRESS can be null.

nud NUD_STATE --- state of the neighbour entry. nud is an abbreviation for "Neighbour Unreachability Detection". This state can take one of the following values:

permanent --- the neighbour entry is valid forever and can be removed only administratively.

noarp --- the neighbour entry is valid, no attempts to validate this entry will be made but it can be removed when its lifetime expires.

reachable --- the neighbour entry is valid until reachability timeout expires.

stale --- the neighbour entry is valid, but suspicious. This option to ip neighbour does not change the neighbour state if the entry was valid and the address has not been changed by this command.

Examples:

ip neigh add 10.0.0.3 lladdr 000001 dev eth0 nud perm

--- add permanent ARP entry for neighbour 10.0.0.3 on the device eth0.

ip neigh chg 10.0.0.3 dev eth0 nud reachable

--- change its state to reachable.

ip neighbour delete --- delete neighbour entry.

Abbreviations: delete, del, d.

This command invalidates a neighbour entry.

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

Example:

ip neigh del 10.0.0.3 dev eth0

--- invalidate ARP entry for neighbour 10.0.0.3 on the device eth0.

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

WARNING!

Attempts to delete or to change manually a noarp entry created by kernel may result in unpredictable behaviour. More specifically the kernel may start trying to resolve this address even on NOARP interfaces or change the address to multicast or broadcast.

ip neighbour show --- list neighbour entries.

Abbreviations: show, list, sh, ls.

This commands displays neighbour tables.

Arguments:

to ADDRESS (default) --- prefix selecting neighbours to list.

dev NAME --- list only neighbours attached to this device.

unused --- list only neighbours, which are not in use now.

nud NUD_STATE --- list only neighbour entries in this state. NUD_STATE takes values listed below after the example or the special value all, which means all the states. This option may occur more than once. If this option

is absent, ip lists all the entries except for none and noarp.

Example:

kuznet@alisa~ $ ip neigh ls

dev lo lladdr 000000000000 nud noarp

fe80200cfffe763f85 dev eth0 lladdr 00000c763f85 router nud stale

0.0.0.0 dev lo lladdr 000000000000 nud noarp

193.233.7.254 dev eth0 lladdr 00000c763f85 nud reachable

193.233.7.85 dev eth0 lladdr 00e01e633900 nud stale

kuznet@alisa~ $

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

lladdr is link layer address of the neighbour.

nud is the state of ``neighbour unreachability detection for this entry. The full list of the possible nud states with minimal descriptions are:

none --- state of the neighbour is void.

incomplete --- the neighbour is in process of resolution.

reachable --- the neighbour is valid and apparently reachable.

stale --- the neighbour is valid, but probably it is already unreachable, so that kernel will try to check it at the first transmission.

delay --- a packet has been sent to the stale neighbour, kernel waits for confirmation.

probe --- delay timer expired, but no confirmation was received. Kernel

has started to probe neighbour with ARP/NDISC messages.

failed --- resolution has failed.

noarp --- the neighbour is valid, no attempts to check the entry will be made.

permanent --- it is noarp entry, but only administrator may remove the entry from neighbour table.

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

IPv6 neighbours can be marked with the additional flag router, which means that that neighbour introduced itself as an IPv6 router.

Option -statistics provides some usage statistics,

kuznet@alisa~ $ ip -s n ls 193.233.7.254

193.233.7.254 dev eth0 lladdr 00000c763f85 ref 5 used 12/13/20 \

nud reachable

kuznet@alisa~ $

Here ref is number of users of this entry, and used is a triplet of time intervals in seconds separated by slashes. The triplet of numbers is coded as {used/confirmed/updated}. In this example they show that

The entry was used 12 seconds ago.

The entry was confirmed 13 seconds ago.

The entry was updated 20 seconds ago.

ip neighbour flush --- flush neighbour entries.

Abbreviations: flush, f.

This commands flushes the neighbour tables. Entries may be selected to flush by various criteria.

This command has the same arguments as show. Note that it will not run when no arguments are given, and that the default neighbour states to be flushed do not include permanent or noarp.

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

netadm@alisa~ # ip -s -s n f 193.233.7.254

193.233.7.254 dev eth0 lladdr 00000c763f85 ref 5 used 12/13/20 \

nud reachable

Round 1, deleting 1 entries

Flush is complete after 1 round

ip route - routing table management.

Abbreviations: route, ro, r.

This command manages the route entries within the kernel routing tables. The kernel routing tables keep information about protocol paths to other networked nodes.

Each route entry has a key consisting of the protocol prefix, which is the pairing of the network address and network mask length, and optionally the Type of Service (TOS) value. An IP packet matches to the route if the highest bits of the packets destination address are equal to the route prefix at least up to the prefix length and if the TOS of the route is zero or equal to TOS of the packet.

If several routes match to the packet, the following pruning rules are used to select the best one:

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

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

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

4. If several routes remain after steps 1-4 have been tried then routes with the best preference value are selected.

5. If we still have several routes then the first of them is selected.

Note the ambiguity of action 5. Unfortunately, Linux historically allowed such a bizarre situation. The sense of the word "the first" depends on the literal order in which the routes were added to the routing table and it is practically impossible to maintain a bundle of such routes in any such order.

For simplicity we will limit ourselves to the case wherein such a situation is impossible and routes are uniquely identified by the triplet of {prefix, tos, preference}. Using the ip command for route creation and manipulation makes it impossible to create such non-unique routes.

One useful exception to this rule is the default route on non-forwarding hosts. It is "officially" allowed to have several fallback routes in cases when several routers are present on directly connected networks. In this case Linux performs "dead gateway detection" as controlled by neighbour unreachability detection and references from the transport protocols to select the working router thus the ordering of the routes is not essential. However in this specific case it is not recommended that you manually fiddle with default routes but instead use the Router Discovery protocol. Actually Linux IPv6 does not even allow user level applications access to default routes.

Of course the route selection steps above are not performed in exactly this sequence. The routing table in the kernel is kept in a data structure which allows achieving the final result with minimal cost. Without depending on any particular routing algorithm implemented in the kernel we can summarize the sequence above as: Route is identified by triplet {prefix,tos,preference} key which uniquely locates the route in the routing table.

Route attributes: Each route key refers to a routing information record. The routing information record contains the data required to deliver IP packets, such as output device and next hop router, and additional optional attributes, such as path MTU or the preferred source address for communicating to that destination.

Route types: It is important that the set of required and optional attributes depends on the route type. The most important route type is a unicast route which describes real paths to another hosts. As a general rule, common routing tables only contain unicast routes. However other route types with different semantics do exist. The full list of types understood by the Linux 2.2 kernel is:

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

unreachable --- these destinations are unreachable; packets are discarded and the ICMP message host unreachable (ICMP Type 3 Code 1) is generated. The local senders get error EHOSTUNREACH.

blackhole --- these destinations are unreachable; packets are silently discarded. The local senders get error EINVAL.

prohibit --- these destinations are unreachable; packets are discarded and the ICMP message communication administratively prohibited (ICMP Type 3 Code 13) is generated. The local senders get error EACCES.

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

broadcast --- the destinations are broadcast addresses, the packets are sent as link broadcasts.

throw --- special control route used together with policy rules. If a throw route is selected then lookup in this particular table is terminated pretending that no route was found. Without any policy routing it is equivalent to the absence of the route in the routing table, the packets are dropped and ICMP message net unreachable (ICMP Type 3 Code 0) is generated. The local senders get error ENETUNREACH.

nat --- special NAT route. Destinations covered by the prefix are considered as dummy (or external) addresses, which require translation to real (or internal) ones before forwarding. The addresses to translate to are selected with the attribute via.

anycast --- (not implemented) the destinations are anycast addresses assigned to this host. They are mainly equivalent to local addresses with the difference that such addresses are invalid to be used as the source address of any packet.

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

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

Actually another routing table always exists which is invisible but even more important. It is the local table (ID 255). This table consists of routes for local and broadcast addresses. The kernel maintains this table automatically and administrators should not modify it and do not even need to look at it in normal operation.

The multiple routing tables come into play when policy routing is used. In policy routing the routing table identifier becomes effectively one more

parameter added to the key triplet {prefix,tos,preference}. Thus under policy routing the route is obtained by {tableid,key triplet} identifing the route uniquely. So you can have several identical routes in different tables that will not conflict as we had mentioned above in in the description of "the first" mechanism.

ip route add --- add new route

ip route change --- change route

ip route replace --- change route or add new one.

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

Arguments:

to PREFIX or to TYPE PREFIX (default) --- destination prefix of the route. If TYPE is omitted, ip assumes type unicast. Another values of TYPE are listed above. PREFIX is IPv4 or IPv6 address optionally followed by slash and prefix length. If the length of the prefix is missing, ip assumes full-length host route. Also there is one special PREFIX --- default --- which is equivalent to IP 0/0 or to IPv6 /0.

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

metric NUMBER or preference NUMBER --- preference value of the route. NUMBER is an arbitrary 32bit number.

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

dev NAME --- the output device name.

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

src ADDRESS --- the source address to prefer using when sending to the destinations covered by route prefix. This address must be defined on a local machine interface. This will come into play when routes and rules are combined with the masquerade rules of the ipchains firewall we discuss later.

realm REALMID --- the realm which this route is assigned to. REALMID may be a number or a string from the file /etc/iproute2/rt_realms.

mtu MTU or mtu lock MTU --- the MTU along the path to destination. If modifier lock is not used, MTU may be updated by the kernel due to Path MTU Discovery. If the modifier lock is used then no path MTU discovery will be performed and all the packets will be sent without the DF bit set for the IPv4 case or fragmented to the MTU for the IPv6 case.

window NUMBER --- the maximal advertised window for TCP to these destinations measured in bytes. This parameter limits the maximal data bursts our TCP peers are allowed to send to us.

rtt NUMBER --- the initial RTT (``Round Trip Time) estimate. Actually, in Linux 2.2 and 2.0 it is not RTT but the initial TCP retransmission timeout. The kernel forgets it as soon as it receives the first valid ACK from peer. Alas, this means that this attribute affects only the connection retry rate and is hence useless.

nexthop NEXTHOP --- nexthop of multipath route. NEXTHOP is a complex value with its own syntax as follows:

via ADDRESS is nexthop router.

dev NAME is output device.

weight NUMBER is weight of this element of multipath route

reflecting its relative bandwidth or quality.

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

protocol RTPROTO --- routing protocol identifier of this route. RTPROTO may be a number or a string from the file /etc/iproute2/rt_protos. If the routing protocol ID is not given ip assumes the protocol is boot. IE. This route has been added by someone who does not understand what they are doing. Several of these protocol values have a fixed interpretation.

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

kernel --- route was installed by the kernel during autoconfiguration.

boot --- route was installed during bootup sequence. If a routing daemon will start, it will purge all of them. This is the value assigned to manually inserted routes that do not have a protocol specified.

static --- route was installed by administrator to override dynamic routing. Routing daemon(s) will respect them and advertise them if it is so configured.

ra --- route was installed by Router Discovery protocol.

Note that the rest of values are not reserved and administrator is free to assign or not assign protocol tags. Routing daemons at least should take care of setting some unique protocol values for themselves such as they are assigned in rtnetlink.h or in the rt_protos database.

onlink --- pretend that the nexthop is directly attached to this link, even if it does match any interface prefix. One application of this option may be found in ip tunnels between dissimilar addresses.

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

Two more commands, prepend and append do exist. Prepend does the same thing as the classic route add command by adding the route even if another route to the same destination already exists. The opposite case is append which adds the route to the end of the list. We strongly reccommend that you avoid using these commands.

Unfortunately, IPv6 currently only understands the append command correctly, all the rest of the set translating to append. Certainly, this will change in the future.

Examples:

Add a plain route to network 10.0.0/24 via gateway 193.233.7.65

ip route add 10.0.0/24 via 193.233.7.65

change it to a direct route via device dummy

ip ro chg 10.0.0/24 via 193.233.7.65 dev dummy

Add default multipath route splitting load between ppp0 and ppp1

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

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

NAT the address 192.203.80.144 to 193.233.7.83 before forwarding

ip route add nat 192.203.80.142 via 193.233.7.83

Note that the reverse NAT translation is setup with policy rules as described in the policy routing section.

ip route delete

Abbreviations: delete, del, d.

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

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

Linux kernel 2.0 had the ability to delete a route selected only by the prefix address while ignoring its netmask. This option does not exist anymore due to the ambiguous nature of the selection. If you wish to have such functionality then look at the ip route flush command which provides a richer set of capabilities.

Examples:

Delete the multipath route created by the add example previously

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

ip route show

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

This format of the command allows viewing the routing tables contents and looking at route(s) as selected by some criteria.

Arguments:

to SELECTOR (default) --- select routes only from the given range of destinations. SELECTOR has optional modifiers (root, match or exact) and

a prefix.

root PREFIX selects routes with prefixes not shorter than PREFIX. IE: root 0/0 selects all the routing table.

match PREFIX selects routes with prefixes not longer than PREFIX. match 10.0/16 selects 10.0/16, 10/8 and 0/0, but it does not select 10.1/16 and 10.0.0/24.

exact PREFIX (or just PREFIX) selects routes exactly with this prefix.

If none of these options are present then the ip command assumes root 0/0 which lists the entire table.

tos TOS or dsfield TOS --- Select only routes with given TOS.

table TABLEID --- Show routes from this table(s). Default setting is to show table main (ID 254). TABLEID may be either ID of a real table or one of the special values:

all --- list all the tables.

cache --- dump routing cache.

IPv6 has only a single table, however splitting into main, local, and cache is emulated by the ip utility.

cloned or cached --- list cloned routes which are routes dynamically forked off of other routes because some route attribute (like MTU) was updated. It is equivalent to table cache.

from SELECTOR --- the same syntax as to SELECTOR but bounds the source address range rather than the destination. Note that the from option only works with cloned routes.

protocol RTPROTO --- list only routes of this protocol.

scope SCOPE_VAL --- list only routes with this scope.

type TYPE --- list only routes of this type.

dev NAME --- list only routes going via this device.

via PREFIX --- list only routes going via selected by PREFIX nexthop routers.

src PREFIX --- list only routes with preferred source addresses selected by PREFIX.

realm REALMID or realms FROMREALM/TOREALM --- list only routes with these realms.

Using this command is best explained by running through some examples.

Example: Let us count the routes of protocol gated/bgp on a router

kuznet@amber~ $ ip route list proto gated/bgp | wc

1413 9891 79010

kuznet@amber~ $

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

kuznet@amber~ $ ip -o route list cloned | wc

159 2543 18707

kuznet@amber~ $

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

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

kuznet@amber~ $ ip route list 193.233.7/24

193.233.7.0/24 dev eth0 proto gated/conn scope link \

src 193.233.7.65 realms inr.ac

kuznet@amber~ $

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

kuznet@amber~ $ ip route list 193.233.7.82 table cache

193.233.7.82 from 193.233.7.82 dev eth0 src 193.233.7.65 \

realms inr.ac/inr.ac

cache <src-direct,redirect> mtu 1500 rtt 300 iif eth0

193.233.7.82 dev eth0 src 193.233.7.65 realms inr.ac

cache mtu 1500 rtt 300

kuznet@amber~ $

This route looks a bit strange, does it not? Did you notice that this is the path from 193.233.7.82 back to 193.233.82? In the section on ip route get you will see how this route is created.

The second line which starts with the word cache shows the additional attributes which normal routes do not possess. The cache flags contained within the angle brackets are:

local --- packets are delivered locally. It stands for loopback unicast routes, for broadcast routes, and for multicast routes if this host is

member of the corresponding group.

reject --- the path is bad. Any attempt to use it results in error. See attribute error below.

mc --- the destination is multicast.

brd --- the destination is broadcast.

src-direct --- the source is on a directly connected interface.

redirected --- the route was created by an ICMP Redirect.

redirect --- packets going via this route will trigger ICMP redirect.

fastroute --- route is eligible to be used for fastroute.

equalize --- make packet by packet randomization along this path.

dst-nat --- destination address requires translation.

src-nat --- source address requires translation.

masq --- source address requires masquerading.

notify --- (not implemented) change/deletion of this route will trigger RTNETLINK notification.

The following are optional attributes that may be present:

error --- on reject routes this is the error code returned to local senders when they try to use this route. These error codes are translated to ICMP error codes sent to remote senders according to the rules described above in the subsection devoted to route types.

expires --- this entry will expire after this timeout.

iif --- the packets for this path are expected to arrive on this interface.

Giving the option -statistics will show further information about this route:

users --- number of users of this entry.

age --- shows when this route was used last time.

used --- number of lookups of this route since its creation.

ip route flush - allows group deletion of routes

Abbreviations: flush, f.

This command allows flushing routes as selected by some criteria.

The arguments have the same syntax and semantics as the arguments of ip route show but the routing tables are purged rather than listed. The only difference is the default action performed. Where the ip route show command dumps the main IP routing table, ip route flush prints the help page. The reason for this difference does not require an explanation does it?

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

Examples:

The first example flushes all the gatewayed routes from main table such as after a routing daemon crash.

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

This option deserved to be put into the scriptlet routef available within the IPROUTE2 utility distribution. This option was described in the route(8) man page as borrowed from BSD but was never implemented in Linux.

The second example is flushing all IPv6 cloned routes:

netadm@amber~ # ip -6 -s -s ro flush cache

3ffe2400220affffef4c5d1 via 3ffe2400220affffef4c5d1 \

dev eth0 metric 0

cache used 2 age 12sec mtu 1500 rtt 300

3ffe2400280adfffeb78034 via 3ffe2400280adfffeb78034 \

dev eth0 metric 0

cache used 2 age 15sec mtu 1500 rtt 300

3ffe2400280c8fffe595bcc via 3ffe2400280c8fffe595bcc \

dev eth0 metric 0

cache users 1 used 1 age 23sec mtu 1500 rtt 300

3ffe2400012a0ccfffe661878 via 3ffe2400012a0ccfffe661878 \

dev eth1 metric 0

cache used 2 age 20sec mtu 1500 rtt 300

3ffe240001a0020fffe71fb30 via 3ffe240001a0020fffe71fb30 \

dev eth1 metric 0

cache used 2 age 33sec mtu 1500 rtt 300

ff021 via ff021 dev eth1 metric 0

cache users 1 used 1 age 45sec mtu 1500 rtt 300

Round 1, deleting 6 entries

Flush is complete after 1 round

netadm@amber~ # ip -6 -s -s ro flush cache

Nothing to flush.

The third example is flushing BGP routing tables after gated death.

netadm@amber~ # ip ro ls proto gated/bgp wc

1408 9856 78730

netadm@amber~ # ip -s ro f proto gated/bgp

Round 1, deleting 1408 entries

Flush is complete after 1 round

netadm@amber~ # ip ro f proto gated/bgp

Nothing to flush.

netadm@amber~ # ip ro ls proto gated/bgp

ip route get - obtain route pathing

Abbreviations: get, g.

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

Arguments:

to ADDRESS (default) --- destination address.

from ADDRESS --- source address.

tos TOS or dsfield TOS --- Type Of Service.

iif NAME --- device, which this packet is expected to arrive from.

oif NAME --- enforce output device, which this packet will be routed out.

connected --- if no source address (option from) was given, relookup the

route with the source address set to the preferred address as received from the first lookup. If policy routing is used this may be a different route.

Note that this operation is not equivalent to ip route show. ip route show shows the existing routes, ip route get resolves them and creates new clones if necessary. Essentially, ip route get is equivalent to actually sending a packet along this path. If the argument iif is not given then the kernel creates a route to output packets towards requested destination. This is equivalent to pinging the destination then running ip route list cache but in the case of ip route get no packets are actually sent. With the argument iif present the kernel pretends that a packet has arrived from this interface and searches for a path to forward the packet. This command outputs routes in the same format as ip route ls.

Examples:

Find route to output packets to 193.233.7.82:

kuznet@amber~ $ ip route get 193.233.7.82

193.233.7.82 dev eth0 src 193.233.7.65 realms inr.ac

cache mtu 1500 rtt 300

kuznet@amber~ $

Find route to forward packets arriving on eth0 from 193.233.7.82 and destined to 193.233.7.82:

kuznet@amber~ $ ip route get 193.233.7.82 from 193.233.7.82 iif eth0

193.233.7.82 from 193.233.7.82 dev eth0 src 193.233.7.65 \

realms inr.ac/inr.ac

cache <src-direct,redirect> mtu 1500 rtt 300 iif eth0

kuznet@amber~ $

This is the operation that created the funny route in the examples to ip route list with 193.233.7.82 looped back to 193.233.7.82. Note the redirect flag present on the output.

Find multicast route for packets arriving on eth0 from host 193.233.7.82 and destined to multicast group 224.2.127.254 assuming that a multicast routing daemon is running such as in this case we are running pimd.

kuznet@amber~ $ ip route get 224.2.127.254 from 193.233.7.82 iif eth0

multicast 224.2.127.254 from 193.233.7.82 dev lo \

src 193.233.7.65 realms inr.ac/cosmos

cache <mc> iif eth0 Oifs eth1 pimreg

kuznet@amber~ $

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

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

netadm@alisa~ # ip route add 193.233.7.98 via 193.233.7.254

netadm@alisa~ # ip route get 193.233.7.98

193.233.7.98 via 193.233.7.254 dev eth0 src 193.233.7.90

cache mtu 1500 rtt 3072

and probe it with ping

netadm@alisa~ # ping -n 193.233.7.98

PING 193.233.7.98 (193.233.7.98) from 193.233.7.90 56 data bytes

From 193.233.7.254 Redirect Host(New nexthop 193.233.7.98)

64 bytes from 193.233.7.98 icmp_seq=0 ttl=255 time=3.5 ms

From 193.233.7.254 Redirect Host(New nexthop 193.233.7.98)

64 bytes from 193.233.7.98 icmp_seq=1 ttl=255 time=2.2 ms

64 bytes from 193.233.7.98 icmp_seq=2 ttl=255 time=0.4 ms

64 bytes from 193.233.7.98 icmp_seq=3 ttl=255 time=0.4 ms

64 bytes from 193.233.7.98 icmp_seq=4 ttl=255 time=0.4 ms

^C

--- 193.233.7.98 ping statistics ---

5 packets transmitted, 5 packets received, 0% packet loss

round-trip min/avg/max = 0.4/1.3/3.5 ms

What occured? The router at 193.233.7.254 understood that we have a much better path to the destination and sent us an ICMP redirect message. We now retry ip route get to see what we have in our routing tables.

netadm@alisa~ # ip route get 193.233.7.98

193.233.7.98 dev eth0 src 193.233.7.90

cache <redirected> mtu 1500 rtt 3072

ip rule --- routing policy database management.

Abbreviations: rule, ru.

Rules in routing policy database controlling route selection algorithm.

Classic routing algorithms used in the Internet make routing decisions based only on the destination address of packets and in theory, but not in practice, on the TOS field. In some circumstances we want to route packets differently depending not only on the destination addresses, but also on other packet fields such as source address, IP protocol, transport protocol ports or even packet payload. This task is called "policy routing".

"policy routing" != "routing policy"

"policy routing" = "cunning routing"

"routing policy" = "routing tactics" or "routing plan"

To solve this task the conventional destination based routing table, ordered according to the longest match rule, is replaced with the "routing policy database" or RPDB, which selects the appropriate route through execution of some set of rules. These rules may have many keys of different natures and therefore they have no natural ordering excepting that which is imposed by the network administrator. In Linux the RPDB is a linear list of rules ordered by a numeric priority value. The RPDB explicitly allows matching packet source address, packet destination address, TOS, incoming interface (which is packet metadata, rather than a packet field), and using fwmark values for matching IP protocols and transport ports.

Each routing policy rule consists of a selector and an action predicate. The RPDB is scanned in the order of increasing priority with the selector of each rule applied to the source address, destination address, incoming interface, tos, and fwmark. If the selector matches the packet the action is performed. The action predicate may return success in which case the rule output provides either a route or a failure indication and RPDB lookup is then terminated. Otherwise, the RPDB program continues on to the next rule.

What is the action semantically? The natural action is to select the nexthop and output device. This is the way a packet path route is selected

by Cisco IOS, let us call it "match & set". In Linux the approach is more flexible as the action includes lookups in destination-based routing tables and selecting a route from these tables according to classic longest match algorithm. The "match & set" approach then becomes the simplest case of Linux route selection realized when the second level routing table contains a single default route. Remember that Linux supports multiple routing tables managed with ip route command.

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

1. Priority 0: Selector = match anything

Action = lookup routing table local (ID 255).

The table local is the special routing table containing high priority control routes for local and broadcast addresses.

Rule 0 is special, it cannot be deleted or overridden.

2. Priority 32766: Selector = match anything

Action = lookup routing table main (ID 254)

The table main is the normal routing table containing all non-policy routes. This rule may be deleted or overridden with other rules.

3. Priority 32767: Selector = match anything

Action = lookup routing table default (ID 253).

The table default is empty and reserved for post-processing if previous default rules did not select the packet. This rule also may be deleted.

Do not mix routing tables and rules. Rules point to routing tables, several rules may refer to one routing table and some routing tables may have no rules pointing to them. If you delete all the rules referring to a table then the table is not used but still exists. A routing table will disappear only after all the routes contained within it are deleted.

Rule attributes: Each RPDB entry has additional attributes attached. Each rule has a pointer to some routing table. NAT and masquerading rules have the attribute to select a new IP address to translate/masquearade. Additionally rules have some of the optional attributes which routes have such as realms. These values do not override those contained in routing tables, they are used only if the route did not select any of those attributes.

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

unicast --- the rule prescribes returning the route found in the routing table referenced by the rule.

blackhole --- the rule prescribes to drop packet silently.

unreachable --- the rule prescribes generating the error "Network is unreachable".

prohibit --- the rule prescribes generating the error "Communication is administratively prohibited".

nat --- the rule prescribes translating the source address of the IP packet to some other value.

ip rule add --- insert new rule

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

Arguments:

type TYPE (default) --- type of this rule. The list of valid types was given in the previous subsection.

from PREFIX --- select source prefix to match.

to PREFIX --- select destination prefix to match.

iif NAME --- select incoming device to match. If the interface is loopback,

the rule matches only packets originated by this host. It means that you may create separate routing tables for forwarded and local packets and, hence, completely segregate them.

tos TOS or dsfield TOS --- select TOS value to match.

fwmark MARK --- select value of fwmark to match.

priority PREFERENCE --- priority of this rule. Each rule should have an explicitly set unique priority value. Priority is an unsigned 32 bit number thus we have 4294967296 possible rules.

WARNING!

For historical reasons ip rule add does not require any priority value and allows the priority value to be non-unique. If the user had not supplied a priority value then one was assigned by the kernel.If the user requested creating a rule with a priority value which already existed then the kernel did not reject the request and added the new rule before all old rules of the same priority. This is a mistake in the current design, nothing more. It should be fixed by the time you read this so please do not rely on this feature. You should always use explicit priorities when creating rules.

table TABLEID --- routing table identifier to lookup if the rule selector matches.

realms FROM/TO --- Realms to select if the rule matched and routing table lookup succeeded. Realm TO is used only if the route returned did not select any realm.

nat ADDRESS --- The base of IP address block to translate source address. The ADDRESS may be either the start of a block of NAT addresses as selected by NAT routes, a local host address, or even zero. In the last two cases the Linux router does not NAT translate the packets but masquerades them to this address.

Changes to the RPDB made with these commands do not become active

immediately. You should run ip route flush cache to flush out the routing cache after inserting rules.

Examples:

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

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

Translate packet source 193.233.7.83 to 192.203.80.144 and route it according to table #1 (Table #1 is defined in /etc/iproute/rt_tables as inr.ruhep)

ip rule add from 193.233.7.83 nat 192.203.80.144 table 1 prio 320

Delete unused default rule

ip rule del prio 32767

ip rule show - list policy rules

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

Good news - this is the only command which has no arguments. Here is the example:

kuznet@amber~ $ ip rule list

0 from all lookup local

200 from 192.203.80.0/24 to 193.233.7.0/24 lookup main

210 from 192.203.80.0/24 to 192.203.80.0/24 lookup main

220 from 192.203.80.0/24 lookup inr.ruhep realms inr.ruhep/radio-msu

300 from 193.233.7.83 to 193.233.7.0/24 lookup main

310 from 193.233.7.83 to 192.203.80.0/24 lookup main

320 from 193.233.7.83 lookup inr.ruhep map-to 192.203.80.144

32766 from all lookup main

In the first position the rule priority value stands followed by a colon. Then the selectors follow with each key prefixed by the keyword used to create the rule.

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

If the rule does NAT, as in rule #320, it is shown by the keyword map-to followed by the start of the block of addresses to map.

The sense of this example is pretty simple. The prefixes 192.203.80.0/24 and 193.233.7.0/24 form an internal network but each prefix is routed differently. Additionally, the host 193.233.7.83 is translated to another prefix as 192.203.80.144 when talking to the outer world.

ip tunnel - ip tunnelling configuration

Abbreviations: tunnel, tunl.

The tunnel objects are tunnels encapsulating packets within IPv4 packets and sending them over the IP infrastructure.

ip tunnel add - creating tunnels

Abbreviations: add, a

Arguments:

name NAME (default) --- select tunnel device name.

mode MODE --- set tunnel mode. Three modes are available: ipip, sit, gre

remote ADDRESS --- set remote endpoint of the tunnel.

local ADDRESS --- set fixed local address for tunneled packets. It must be

an address on another interface of this host.

ttl N --- set fixed TTL N on tunneled packets. N is number in the range 1--255. 0 is special value, meaning that packets inherit TTL value. Default value is inherit.

tos TOS or dsfield TOS --- set fixed TOS on tunneled packets. Default value is inherit.

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

nopmtudisc --- disable Path MTU Discovery on this tunnel. It is enabled by default. Note that a fixed ttl is incompatible with this option. A tunnel with fixed ttl always performs pmtu discovery.

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

csum, icsum, ocsum --- (GRE only) checksum tunneled packets. The flag ocsum orders checksumming outgoing packets, icsum requires that all the input packets have a correct checksum. csum is equivalent to the combination "icsum ocsum".

seq, iseq, oseq --- (GRE only) serialize packets. The flag oseq enables sequencing outgouing packets, iseq requires that all the input packets were serialized. seq is equivalent to the combination "iseq oseq".

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

Examples:

Create pointopoint IPv6 tunnel with maximal TTL of 32.

ip tunl add Cisco mode sit remote 192.31.7.104 local 192.203.80.142 ttl 32

ip tunnel show - list tunnel attributes

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

Example:

kuznet@amber~ $ ip tunl ls Cisco

Cisco: ipv6/ip remote 192.31.7.104 local 192.203.80.142 ttl 32

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

kuznet@amber~ $ ip -s tunl ls Cisco

Cisco ipv6/ip remote 192.31.7.104 local 192.203.80.142 ttl 32

RX Packets Bytes Errors CsumErrs OutOfSeq Mcasts

12566 1707516 0 0 0 0

TX Packets Bytes Errors DeadLoop NoRoute NoBufs

13445 1879677 0 0 0 0

Essentially these numbers are the same as those printed using ip -s link show but the tags are different to reflect tunnel specific features. These features are:

CsumErrs --- total number of packets dropped because of checksum failures for GRE tunnel with enabled checksumming.

OutOfSeq --- total number of packets dropped because they arrived out of sequence for GRE tunnel with enabled serialization.

Mcasts --- total number of multicast packets, received on broadcast GRE

tunnel.

DeadLoop --- total number of packets, which were not transmitted because tunnel is looped back to itself.

NoRoute --- total number of packets, which were not transmitted because there is no IP route to remote endpoint.

NoBufs --- total number of packets, which were not transmitted because kernel failed to allocate buffer.

ip monitor and rtmon --- route state monitoring

The ip utility allows monitoring the state of devices, addresses, and routes continuously. This option has a different format in that the command monitor is first on the command line followed by the object list.

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

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

If a file name is given ip does not listen to RTNETLINK but opens the file which is assumed to contain RTNETLINK messages saved in binary format and dumps them. Such a history file can be generated with the utility rtmon. This utility has a command line syntax similar to ip monitor. Ideally, rtmon should be started before the first network configuration command is issued. It is possible to start rtmon at any time as it prepends the history with the system state snapshot dumped at the moment of startup.

rtacct - route realms and policy propagation

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

source, the task presents a quadratic complexity and its solution is physically impossible.

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

Within the Linux architecture where we have a split RIB as maintained by user level daemon, and the kernel based FIB (Forwarding Information Base), we cannot allow such a simplistic approach.

Fortunately there exists another solution allowing an even more flexible policy with rich semantics. Routes can be clustered together in user space based on their attributes. IE: A BGP router knows the route ASPATH or its community whereas an OSPF router knows the route tag or its area. A network administrator adding routes manually knows the nature of those routes. Providing that the number of such aggregates, which we call realms, is low, the task of full classification both by source and destination becomes quite manageable.

So each route may be assigned to a realm. It is assumed that this identification is made by a routing daemon, but static routes may also be assigned manually through ip route.

Currently there exists a patch to gated allowing it to classify routes to realms over all the set of policy rules. This classification is implemented within gated by prefix, ASPATH, origin, tag, etc.

To facilitate this construction in the case when the routing daemon is not aware of realms, missing realms may be completed with routing policy rules.

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

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

2. If rule has a source realm, source realm of the packet is set to it.

3. If destination realm was not obtained from route and rule has destination realm, set destination realm from rule.

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

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

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

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

The main application of realms is in conjunction with the tc route classifier where they are used to help assign packets to traffic classes, for accounting, policing, and scheduling them according to the classification.

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

kuznet@amber~ $ rtacct russia

Realm BytesTo PktsTo BytesFrom PktsFrom

russia 20576778 169176 47080168 153805

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

Note that locally originated packets are not accounted here as rtacct

shows ingoing packets only. Using the route classifier you can get even more detailed accounting information about outgoing packets, optionally summarizing traffic not only by source or destination, but by any pair of source and destination realms.

IP Utility Summary

We have presented in this section coverage of the ip utility from the IPROUTE2 utility suite. As we have shown this is the replacement under Linux for the ifconfig and route utilities for performing advanced IP network manipulation. While the standard utilities will suffice for simple setups we recommend using the ip utility instead in order to both be familiar with the usage as well as able to utilize the vast power of this utility. Linux possesses one of the most complete and powerful implementations of IP networking facilities available. We will now cover some of the basics of using the ip utility within scripts.

IP Usage in Scripting

In this section we will use what we have learned about the ip utility to create and learn from several scripts. First we will create ipup and ipdown scripts for our system. Then we will cover the operation of Alexey's ifcfg script from IPROUTE2 that uses the ip utilities to provide a stronger version of ifconfig. Finally we will cover an example of creating multiple route tables for splitting up outgoing traffic.

IPUP & IPDOWN

In this section we will create some custom networking scripts along with the core /etc/rc.d/init.d/network script using the IPROUTE2 utility suite.

First let us consider how we would manually configure the interfaces with the ip utility. The first interface is lo and it was configured under ifconfig as: ifconfig lo 127.0.0.1 netmask 255.0.0.0 broadcast 127.255.255.255. Rewriting this in ip we get two lines because of the granularity of control. So we have:

ip addr add 127.0.0.1/8 dev lo broadcast +

ip link set lo up

If we want to substitute this directly into the ipup script we would fail as the format of the configuration file, ifcfg-lo, is different from the information we need to configure the interface using ip. Also remember that the ip utility allows us to set multiple addresses on a single interface and our ipup script should allow us to take full advantage of that facility without requiring it.

To configure multiple address on our network interface we will use a loop over all the possible values of addresses within a variable. Additionally we may want to allow for renaming the device before assigning addresses so that the output of our listings makes better logical sense. So first we should think about what variables we would require in a configuration script. Then we can start writing the ipup script to take advantage of the ip utility functions and our configuration variables.

Consider the following interface configuration file:

#!/bin/sh

#>>>Device type: ethernet

#>>>Variable declarations:

DEVICE=eth0

DEVNAME=inet0

IPCIDR="192.168.1.1/24

10.3.123.1/28"

STARTME=1

#>>>End variable declarations

We have ip addresses recorded in the CIDR format that is used by the ip addr command. We have the actual kernel boot supplied interface name and also a variable for renaming the device. Finally we have the on-boot initialization switch. We want our ipup script to allow on-boot init as well as after boot init functions. Since we can define more than one IPv4 address within this configuration we need a loop function to iterate the address assignment. Combining all of these needs we get the following script.

Begin Listing - ipup script

```
#!/bin/bash

cd /etc/sysconfig/network-scripts/

. $1

if ([ $STARTME -eq 1 ] || [ "$2" = "now" ])

then

/sbin/ip link set $DEVICE down

DEV=$DEVICE

if [ -n $DEVNAME ]; then

/sbin/ip link set $DEVICE name $DEVNAME

DEV=$DEVNAME

fi

for ADDRESS in $IPCIDR

do

/sbin/ip addr add $ADDRESS broadcast + dev $DEV
```

done

/sbin/ip link set $DEV arp on

/sbin/ip link set $DEV up

fi

End Listing

Note that we allow for both changing or not changing the device name. The inner loop assigns all addresses that are listed in the IPCIDR variable to the device. Thus with a simple config file and a short ipup script we can setup our network devices with custom names and multiple addresses.

Let us take a quick look at the related ipdown script that uses the ip utility.

Begin Listing - ipdown script

#!/bin/bash

cd /etc/sysconfig/network-scripts/

. $1

DEV=$DEVICE

if [-n $DEVNAME]; then

DEV=$DEVNAME

fi

for ADDRESS in $IPCIDR

do

/sbin/ip addr del $ADDRESS dev $DEV

done

/sbin/ip link set $DEV down

/sbin/ip link set $DEV arp off

if [-n $DEVNAME]; then

ip link set $DEVNAME name $DEVICE

fi

End Listing

Note that we change the device name back to the original kernel defined name. That way we can switch between using any set of utilities we want as any particular set will restore the device to the same state as it started from.

IPNetwork Init Script

Now that we have new ipup, ipdown, and ipcfg-xxx files, let us turn our attention to the init file that runs the ipup script on system bootup. On our systems this file is called ipnetwork and resides in the /etc/rc.d/init.d/ directory. We will consider the final format of this file as it is written with the IPROUTE2 utilities in mind.

Begin Listing - /etc/rc.d/init.d/ipnetwork

#

IP network Turn on/off IP networking

#

Source function library.

. /etc/rc.d/init.d/pakinit.functions

. /etc/sysconfig/ipnetwork

```
cd /etc/sysconfig/network-scripts

# See how we were called.

case "$1" in

start)

pakcmd "ipup.mon" "Starting Monitor" exec /sbin/rtmon file \

/var/log/iproute.log &

pakcmd "ipup.lo" "Starting LoopBack " ./ipup ipcfg-lo

for IF in $INTERFACES ; do

for i in ipcfg-$IF[0-9]; do

pakcmd "ipup.$i" " Starting IP Interface $i " ./ipup $i

done

if [ -r ipcfg-routes ]; then

pakcmd "ipup.2" "Starting IP Static Routes " ./ipcfg-routes

fi

done

;;

stop)

for IF in $INTERFACES; do

for i in ipcfg-$IF[0-9]; do

pakcmd "ipdown.$i" "Downing IP Interface $i " ./ipdown $i

done
```

done

;;

*)

echo "Usage: ipnetwork {start|stop}"

exit 1

esac

exit 0

End Listing

We now posess a complete set of IP configuration scripts that will use the ip utility to create and destroy IP interfaces. Note that when we consider IPv6 these scripts can also be used with only minor changes. Then we will have configuration files and scripts that cover both protocols.

ifcfg script

We will now dissect a shell script provided in the IPROUTE2 package. This script is called ifcfg and Alexey wrote it as a replacement for ifconfig. Here is the full text of the script:

Begin Listing - ifcfg script

#! /bin/bash

CheckForwarding () {

local sbase fwd

sbase=/proc/sys/net/ipv4/conf

fwd=0

if [-d $sbase]; then

```
for dir in $sbase/*/forwarding; do

fwd=$[$fwd + `cat $dir`]

done

else

fwd=2

fi

return $fwd

}

RestartRDISC () {

killall -HUP rdisc || rdisc -fs

}

ABCMaskLen () {

local class;

class=${1%%.*}

if [ "$1" = "" -o $class -eq 0 -o $class -ge 224 ]; then return 0

elif [ $class -ge 224 ]; then return 0

elif [ $class -ge 192 ]; then return 24

elif [ $class -ge 128 ]; then return 16

else return 8; fi

}

label="label $1"
```

```
ldev="$1"

dev=${1%:*}

if [ "$dev" = "" -o "$1" = "help" ]; then

echo "Usage: ifcfg DEV [[add|del [ADDR[/LEN]] [PEER] | stop]" 1>&2

echo " add - add new address" 1>&2

echo " del - delete address" 1>&2

echo " stop - completely disable IP" 1>&2

exit 1

fi

shift

CheckForwarding

fwd=$?

if [ $fwd -ne 0 ]; then

echo "Forwarding is ON or its state is unknown ($fwd). OK, No RDISC."
1>&2

fi

deleting=0

case "$1" in

add) shift ;;

stop)

if [ "$ldev" != "$dev" ]; then
```

```
echo "Cannot stop alias $ldev" 1>&2

exit 1;

fi

ip -4 addr flush dev $dev $label || exit 1

if [ $fwd -eq 0 ]; then RestartRDISC; fi

exit 0 ;;

del*)

deleting=1; shift ;;

*)

esac

ipaddr=

pfxlen=

if [ "$1" != "" ]; then

ipaddr=${1%/*}

if [ "$1" != "$ipaddr" ]; then

pfxlen=${1#*/}

fi

if [ "$ipaddr" = "" ]; then

echo "$1 is bad IP address." 1>&2

exit 1

fi
```

```
fi

shift

peer=$1

if [ "$peer" != "" ]; then

if [ "$pfxlen" != "" -a "$pfxlen" != "32" ]; then

echo "Peer address with non-trivial netmask." 1>&2

exit 1

fi

pfx="$ipaddr peer $peer"

else

if [ "$pfxlen" = "" ]; then

ABCMaskLen $ipaddr

pfxlen=$?

fi

pfx="$ipaddr/$pfxlen"

fi

if [ "$ldev" = "$dev" -a "$ipaddr" != "" ]; then

label=

fi

if [ $deleting -ne 0 ]; then

ip addr del $pfx dev $dev $label || exit 1
```

```
if [ $fwd -eq 0 ]; then RestartRDISC; fi

exit 0

fi

if ! ip link set up dev $dev ; then

echo "Error: cannot enable interface $dev." 1>&2

exit 1

fi

if [ "$ipaddr" = "" ]; then exit 0; fi

if ! arping -q -c 2 -w 3 -D -I $dev $ipaddr ; then

echo "Error: some host already uses address $ipaddr on $dev." 1>&2

exit 1

fi

if ! ip address add $pfx brd + dev $dev $label; then

echo "Error: failed to add $pfx on $dev." 1>&2

exit 1

fi

arping -q -A -c 1 -I $dev $ipaddr

noarp=$?

( sleep 2 ;

arping -q -U -c 1 -I $dev $ipaddr ) >& /dev/null </dev/null &

ip route add unreachable 224.0.0.0/24 >& /dev/null
```

```
ip route add unreachable 255.255.255.255 >& /dev/null

if [ `ip link ls $dev | grep -c MULTICAST` -ge 1 ]; then

ip route add 224.0.0.0/4 dev $dev scope global >& /dev/null

fi

if [ $fwd -eq 0 ]; then

if [ $noarp -eq 0 ]; then

ip ro append default dev $dev metric 30000 scope global

elif [ "$peer" != "" ]; then

if ping -q -c 2 -w 4 $peer ; then

ip ro append default via $peer dev $dev metric 30001

fi

fi

RestartRDISC

fi

exit 0
```

End Listing

We will take this script apart piece by piece and explain what it is doing. At the end of this you should have a good understanding of the way an IP address can be checked for correct operation.

First off notice that there are several functions defined early in the script. The first one, CheckForwarding(), performs a check using the integer values present within the interface forwarding sysctl. The second one, RestartRDISC(), is for restarting the router discovery daemon. The third

one, ABCMasqLen(), is just for making assumptions about the standard class netmask.

The script begins by assigning the first argument as the device name and then performing some error checking. If the arguments are incorrect or the help switch was provided then the usage for the command is printed out. Note from the usage statement that this command expects the netmask to be in CIDR format. If the netmask is not in CIDR format or is provided incorrectly then the class assumption is made. Having assigned the device name to the ldev variable we shift the arguments and check on our forwarding setup. If the forwarding is on we print a message and continue assigning the forwarding result to the fwd variable.

We next take up the case statement that determines what operations we will perform on the interface. In the case of add we shift arguments and continue, in the case of del we set a variable then shift and continue. The case of stop brings up a quick flush of the entire interface ip addressing. Note that on the stop routine we first check to make sure we are not dealing with legacy aliased devices. Such devices use the dev:# format and should not be used anymore due to the new multiple address structure available for IP. Note also that after we flush the interface addressing we restart the router discovery daemon if our forwarding sysctl is equal to zero. If we are running a router then we will have set the forwarding status to ensure that other devices can interoperate with us. See Chapter 4 on sysctl for more information.

In the case we are adding or deleting an interface address we continue on through checking the given address and mask length. Then we check on the peer address and determine if it is a single valued ip address. Once we pass these checks we test the netmask to determine if we can safely use it. If the netmask does not exist then we call the standard class netmask function to determine the standard class for the given ip address. This function will return the class netmask as a CIDR mask value based solely on the first octet of the address. Once we either have a defined netmask or have generated one from our address we then can define our ip

address completely using CIDR format.

We have now completed parsing our arguments and now start into the actual work of manipulating the interface. We first cover the case where we are deleting the ip address from the interface. After deleting the address we again restart the router discovery daemon if our forwarding sysctl is equal to zero to ensure the update of the routing tables. If we are not deleting an address from an interface we start running the verification testing. This is where we can learn how to create better scripts for our own interface addressing.

The first test run simply verifies that the interface device can be enabled. If not then the script aborts because without a running device we cannot do any configuration of the addressing. After determining that the ip address we will use is non-null we run a duplicate address test. This is an important check to see if the address we want to use already exists on the local network. This uses the arping utility which can manipulate arp functions. This utility is very powerful and provides quite a few functions for determining and using the ip network structure. We will diverge a bit here to discuss this utility.

arping utility

The arping utility is one of several helpful ip utilities provided in the iputils package from Alexey Kuznetsov. The utilities in this package include arping, clockdiff, ping, ping6, rdisc, tracepath, tracepath6, and traceroute6. The collection should be installed on any machine where you will be running any of the advanced ip networking functions. These commands can be used to disrupt the network so caution must be exercised in their use and accessability.

Arping itself provides an IPv4 ping utility that uses ARP packets for communication. This is very usefull for manipulating arp tables on other local network devices. The arping utility can provide duplicate address testing on the local network and two types of unsolicited ARP output to enable quick updating of local network device arp tables. This latter

functionality can be used to create hot standby servers on a network that allow failover of identical ip addresses to alternate devices. It can also be used to wreak havok on a local network that is not configured to prevent sabatoge.

WARNING!

If you do not understand how these functions work then we strongly reccommend that you obtain a copy of TCP/IP Illustrated Vol. 1 by W. Richard Stevens and read it. Without a firm understanding of the basic mechanisms of TCP/IP v.4 network communication most of the utilities we discuss and procedures we execute can cause severe disruption of your network.

Now that we understand what this utility provides let us return to the discussion of the ifcfg script.

We have now checked that the ip address provided on the command line is not already in use on our local networks. We now assign the ip address to the associated device and determine the correct completion of the command. Once this assignment has succesfully completed we use the unsolicited ARP mechanism of arping to update the arp caches on all of the neighboring devices. This provides instant access to our ip device from any of the local network ip devices.

WARNING!

If you have Win95/98/NT machines in your network be warned that the Windows TCP/IP network stack performs duplicate address testing incorrectly as specified by RFC-2131. Windows TCP/IP stack sends out a gratuitous ARP immediately on starting TCP/IP. This forces interruption of IP networking services if the IP address is already in use and prevents the TCP/IP stack on Windows from starting. There is no known way to workaround this fatal bug in Windows TCP/IP stacks. Do not use proxy ARP or the arping functions in a Windows TCP/IP environment.

Sadly enough the arping utility is very popular among disgruntled network

people as an anonymous way of preventing NT servers from starting up. It is ridiculously simple to have the Linux machine watch for GratArp requests and issue an arping response thus preventing the NT from enabling the network card. And since ARP must be specially watched for by almost all network management systems it is rarely detected that this trick is being played. And even if you are watching you REALLY need to know what is going on as it looks "normal" for the original transaction to take place.

Quoting Alexey here as he replied to a MCSE who brought this up on LinNetDev:

"You have learned your networking from a broken pile of crap and you expect me to break my system so that you in your dumbness will be happy?"

Of course I almost did not include this here as there are enough problems in this world without purposefully baiting the stupid.. 8-} - Enough said.

Now that we have installed our address and updated the local network hosts we turn to setting up a corrected routing structure. We will first deal with routes to the multicast address class and the broadcast class. We start by sending both of these routes into the table main with a setting of host unreachable. We then test our link device for multicast capabilities. If the link is multicast enabled we allow the route for the multicast address class to be assigned to the device.

Finally we test again if the interface is forwarding. In the case of no forwarding we further test for the arp capabilities and peer addresses. If our interface has arp capabilities then we place a default route with a high metric out our device. In the case where we have a peer address then we test for the presence and insert a default route via our peer address with a somewhat high metric. In either case with no forwarding we restart router discovery as the final step.

Now that we have covered the script operation let us look at the utilities

and logic behind it with an eye towards modifying our own interface address assignment scripts. First of all we will note the use of the router discovery agent. This agent is one of the reasons we stressed in the parts on ip route why you should always add a protocol level to your routes. We stressed that if you will be using ip route to add routes to the tables that you code them with protocol static to enable the kernel to know that they are valid static routes. Here is one of the reasons why this is important. Under router discovery the rdisc daemon can override routes that are non protocol tagged. So if we had just placed a default route into our table and we then start router discovery we will find that our route is not being used unless we coded the protocol. This is even more important if we will use any of the routing daemons such as zebra or gated.

Note that even in this script we need to try and use CIDR notation format for our IP addressing. This is actually a very good requirement as it speaks directly to the function of address masking. IP address masks, and IPX address masks as well, require that the mask portion be contiguous. When we write out a mask using the old style dotted decimal it is impossible to indicate the continuity of the mask. Consider the address mask 255.252.255.0. If you do not understand that masks must be contiguous then this looks like it could be a valid mask. We say that knowing that many people configuring IP systems rely on the numbers belonging to the set of good numbers. This set is: 255,254,252,248,240,224,192,128 So the assumption is that if these numbers are present then the mask must be valid. Using CIDR style notation we indicate the number of contiguous ones starting from the left in the mask. In this manner it is impossible to specify an invalid mask. Additionally you can readily see the scope of the address mask in CIDR notation thus making it easier to see where a route would be a more specific or general set of another route. So our choice of using CIDR address notation within our configuration file turns out to be the best way of specifying our addressing.

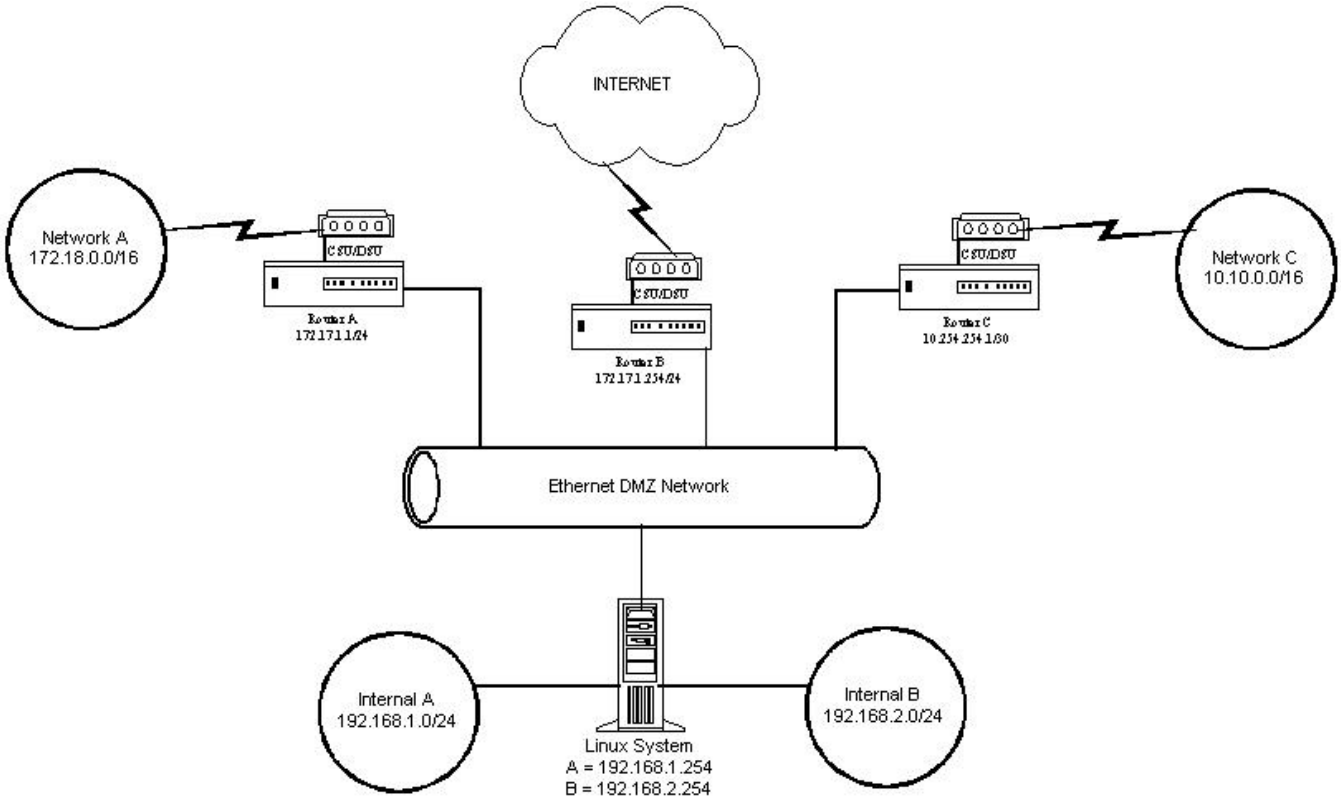
As far as considering the uses of the arping utility, as we mentioned in the warning above Microsoft Windows IPv4 stacks do not handle duplicate

address detection correctly thus using arping on such a network can be problematic. We would not recommend using it except for the duplicate address detection. In our case we will not want to place it into our scripts except in cases where we know that we will need to perform duplicate address detection due to oddly configured DHCP servers and other such high levels of keyboard-seat interface errors.

Policy Routing - Multiple Route Tables Example

We will now consider what is arguably the most powerful feature in the Linux kernel routing code: The use of multiple routing tables combined with policy based routing. In the following text we will present an example of a system acting as a router for three disparate networks. We will return to this example and consider the ipchains utility in the latter half of this book where we cover Linux security and firewalls.

First off look at the diagram of the network we have under consideration:



Multiple Route Tables Network Diagram

Note that we have three external networks attached to our external ethernet interface. Each of these networks has their own router and their

own IP address space that we need to use. Note that two of these address spaces overlap thus adding in a degree of complexity. We will want to setup our routing tables to allow the following connectivity.

* All traffic from any internal network may go to the Internet

* Traffic from Internal B may go to Network A

* Traffic from Internal A may go to Network C

* Traffic from Internal A Hosts 33-62 may go to Network A

* Traffic from Internal B Hosts 65-78 may go to Network C

First we will want to setup our external IP addressing. We will setup two addresses on our DMZ ethernel interface.

Begin Listing

ip addr add 10.254.254.2/30 dev eth0

ip addr add 172.17.1.128/24 dev eth0

End Listing

Next we will cover what route tables we will want to create.

One of the best ways to look at this is to consider that policy routing enables us to determine what routing table to use for source addresses. So the rules should enable us to segment the internal networks. Then we can setup normal destination based routes within the tables. So let us create two new route tables and then discuss the ramifications of this decision.

First we will name the tables by editing the /etc/iproute2/rt_tables file. We will end up with a file that looks like the following:

Begin Listing

#

reserved values

#

255 local

254 main

253 default

0 unspec

#

Local Tables

#

1 networka

2 networkb

End Listing

Now we can refer to these tables in the rule commands. First we will set up the routes in each of these tables.

When you go about setting up the routes within tables there is an approach you can take which will help clarify the steps. Imagine that you are configuring a router that only has two interfaces. The outgoing interface attaches to any outbound router and the incoming interface already only has the packets you want to route. Then it is simply a matter of setting up the routing that you would want within that scope. Let us illustrate by running through the setup for table 1, networka. Here are the commands we would need to input to get the networka routing table configured:

create networka routing table

ip route add 10.10.0.0/16 via 10.254.254.2 table networka proto static

ip route add default via 172.17.1.254 table networka proto static

End

Now what is the command sequence for table networkb? Exactly.

create networkb routing table

ip route add 172.18.0.0/16 via 172.17.1.1 table networkb proto static

ip route add default via 172.17.1.254 table networkb proto static

End

Let us step back a moment and discuss why we have only two tables with three destinations. Notice that we have duplicated the destination default for the Internet into both tables. Why not place this into a third table that we will refer to? The best way to answer this is to consider the interaction between the rules and the tables. Remember that the rules define the policy based routing structure. Multiple rules may point to the same table. However once you are in a table you will need to either obtain a route or be returned via a throw route to the rule list. So if we have matched a rule we would like to assume that we have a correct match of our policy. Thus we would like all possible routes for that packet to be present in the routing table to which the packet is sent. In the case where we have three tables then we would have to have additional rules that actually need to look at the destination of the packet. But looking at the packet destination is the function of a standard route. So why have rules for every possible combination of source AND destination? By using the table we have we can create a few rules that will serve our purpose. Of course the flexibility of the system allows doing the other way around or even through granular inspection of the packets themselves. You should work through all of these scenarios for yourself and decide what works best for you. Enough

theorizing, onwards to the action.

ip rule set #1

ip rule add from 192.168.1.32/27 to 172.18.0.0/16 pref 15000 table networka

ip rule add from 192.168.2.64/28 to 10.10.0.0/16 pref 15001 table networkb

ip rule add from 192.168.1.0/24 pref 15002 table networkb

ip rule add from 192.168.2.0/24 pref 15003 table networka

End

Note that we have used the preference settings to run our rules from most detailed to most general. Remember from the discussion of ip rule that there are two default rules with higher priorities present to catch whatever we do not specify here. These two default rules are very important. Think about what would happen if we forgot about those rules and specified priorities for our rules such as 65535? Would our rules ever be used?

So now consider what will happen to a packet incoming from one of our internal networks. First it will be passed through the rule priority 0 which will pass on it. Then it hits rule priority 15000. If it matches it will be routed according to table networka. If not it runs through rule 15001, then 15002, and finally 15003. Will such a packet ever continue on beyond rule priority 15003?

Now let us confuse the issue. We will redo our tables and rules from a different angle just to illustrate the range of flexibility we have to specify our routing structure.

First let us provide some details about our Linux server. Our Linux server has the following network interfaces and addresses:

eth0 - DMZ ethernet - addresses: 10.254.254.2/30, 172.17.1.128/24

eth1 - Internal A - addresses: 192.168.1.254/24

eth2 - Internal B - addresses: 192.168.2.254/24

Now we will run through the route and rule creation assuming we are starting from the beginning. First edit /etc/iproute2/rt_tables:

#/etc/iproute2/rt_tables

reserved values

#

255 local

254 main

253 default

0 unspec

#

Local Tables

#

1 int1

2 int2

#

Create the routes and rules.

ip rule set #2

ip route add 10.10.0.0/16 via 10.254.254.1 table int1 proto static

ip route add throw 0/0 table int1 proto static

ip route add 172.18.0.0/16 via 172.17.1.1 table int2 proto static

ip route add throw 0/0 table int2 proto static

ip route add 0/0 via 172.17.1.254 table main proto static

ip rule add pref 15000 table int1 iif eth1

ip rule add pref 15001 table int2 iif eth2

ip rule add pref 15002 to 10.10.0.0/16 table int1

ip rule add pref 15003 to 172.18.0.0/16 table int2

End

This set of routes and rules will perform the same operations as set #1. Study them until you see why. Hint: Do not forget the default rules. Later we will see how to use policy routing to perform miraculous tricks with packet paths.