iproute2 Cheat Sheet

by TME520 (TME520) via cheatography.com/20978/cs/4067/

Address management

In this section ${address} value should be a host address in dotted decimal format, and ${mask} can be either a dotted decimal subnet mask or a prefix length. That is, both 192.0.2.10/24 and 192.0.2.10/255.255.255.0 are equally acceptable.

Show all addresses	*ip address show*	All "show" commands can be used with "-4" or "-6" options to show only IPv4 or IPv6 addresses.	
Show addresses for a single interface	*ip address show ${interface name}*	*ip address show eth0*	
Show addresses only for running interfaces	*ip address show up*		
Show only statically configured addresses	*ip address show [dev ${interface}] permanent*		
Show only addresses learnt via autoconfiguration	*ip address show [dev ${interface}] dynamic*		
Add an address to an interface	*ip address add ${address}/${mask} dev ${interface name}*	*ip address add 192.0.2.10/27 dev eth0*	*ip address add 2001:db8:1::/48 dev tun10*
You can add as many addresses as you want. The first address will be primary and will be used as source address by default.			
Add an address with human-readable description	*ip address add ${address}/${mask} dev ${interface name} label ${interface name}:${description}*	*ip address add 192.0.2.1/24 dev eth0 label eth0:my_wan_address*	Interface name with a colon before label is required, some backwards compatibility issue.
Delete an address	*ip address delete ${address}/${prefix} dev ${interface name}*	*ip address delete 192.0.2.1/24 dev eth0*	Interface name argument is required. Linux does allow to use the same address on multiple interfaces and it has valid use cases.
Remove all addresses from an interface	*ip address flush dev ${interface name}*	*ip address flush dev eth1*	

Metasyntactic variables are written in shell-style syntax, ${something}. Optional command parts are in square brackets. Note that there is no way to rearrange addresses and replace the primary address. Make sure you set the primary address first.

Route management

View all routes	*ip route*	*ip route show*
View IPv6 routes	*ip -6 route*	
View routes to a network and all its subnets	*ip route show to root ${address}/${mask}*	*ip route show to root 192.168.0.0/24*
View routes to a network and all supernets	*ip route show to match ${address}/${mask}*	*ip route show to match 192.168.0.0/24*
View routes to exact subnet	*ip route show to exact ${address}/${mask}*	*ip route show to exact 192.168.0.0/24*

Route management (cont)

View only the route actually used by the kernel	*ip route get ${address}/${mask}*	*ip route get 192.168.0.0/24*	Note that in complex routing scenarios like multipath routing, the result may be "correct but not complete", as it always shows one route that will be used first.
View route cache (pre 3.6 kernels only)	*ip route show cached*	Until the version 3.6, Linux used route caching. In older kernels, this command displays the contents of the route cache. It can be used with modifiers described above. In newer kernels it does nothing.	
Add a route via gateway	*ip route add ${address}/${mask} via ${next hop}*	*ip route add 192.0.2.128/25 via 192.0.2.1*	*ip route add 2001:db8:1::/48 via 2001:db8:1::1*
Add a route via interface	*ip route add ${address}/${mask} dev ${interface name}*	*ip route add 192.0.2.0/25 dev ppp0*	Interface routes are commonly used with point-to-point interfaces like PPP tunnels where next hop address is not required.
Change or replace a route	*ip route change 192.168.2.0/24 via 10.0.0.1*	*ip route replace 192.0.2.1/27 dev tun0*	
Delete a route	*ip route delete ${rest of the route statement}*	*ip route delete 10.0.1.0/25 via 10.0.0.1*	*ip route delete default dev ppp0*
Default route	*ip route add default via ${address}/${mask}*	*ip route add default dev ${interface name}*	*ip -6 route add default via 2001:db8::1*
Blackhole routes	*ip route add blackhole ${address}/${mask}*	*ip route add blackhole 192.0.2.1/32*	Traffic to destinations that match a blackhole route is silently discarded.
Other special routes : unreachable	*ip route add unreachable ${address}/${mask}*		Sends ICMP "host unreachable". These routes make the system discard packets and reply with an ICMP error message to the sender.
Other special routes : prohibit	*ip route add prohibit ${address}/${mask}*		Sends ICMP "administratively prohibited".

Route management (cont)

Other special routes : throw	*ip route add throw ${address}/${mask}*		Sends "net unreachable".
Routes with different metric	*ip route add ${address}/${mask} via ${gateway} metric ${number}*	*ip route add 192.168.2.0/24 via 10.0.1.1 metric 5*	*ip route add 192.168.2.0 dev ppp0 metric 10*
Multipath routing	*ip route add ${addresss}/${mask} nexthop via ${gateway 1} weight ${number} nexthop via ${gateway 2} weight ${number}*	*ip route add default nexthop via 192.168.1.1 weight 1 nexthop dev ppp0 weight 10*	

As per the section below, if you set up a static route, and it becomes useless because the interface goes down, it will be removed and never get back on its own. You may not have noticed this behaviour because in many cases additional software (e.g. NetworkManager or rp-pppoe) takes care of restoring routes associated with interfaces.

Link management

Show information about all links	*ip link show*	*ip link list*	
Show information about specific link	*ip link show dev ${interface name}*	*ip link show dev eth0*	*ip link show dev tun10*
Bring a link up or down	*ip link set dev ${interface name} [up | down]*	*ip link set dev eth0 down*	*ip link set dev br0 up*
Set human-readable link description	*ip link set dev ${interface name} alias "${description}"*	*ip link set dev eth0 alias "LAN interface"*	
Rename an interface	*ip link set dev ${old interface name} name ${new interface name}*	*ip link set dev eth0 name lan*	Note that you can't rename an active interface. You need to bring it down before doing it.
Change link layer address (usually MAC address)	*ip link set dev ${interface name} address ${address}*	*ip link set dev eth0 address 22:ce:e0:99:63:6f*	
Change link MTU	*ip link set dev ${interface name} mtu ${MTU value}*	*ip link set dev tun0 mtu 1480*	
Delete a link	*ip link delete dev ${interface name}*		
Enable or disable multicast on an interface	*ip link set ${interface name} multicast on*	*ip link set ${interface name} multicast off*	
Enable or disable ARP on an interface	*ip link set ${interface name} arp on*	*ip link set ${interface name} arp off*	

Link management (cont)

Create a VLAN interface	*ip link add name ${VLAN interface name} link ${parent interface name} type vlan id ${tag}*	*ip link add name eth0.110 link eth0 type vlan id 110*	The only type of VLAN supported in Linux is IEEE 802.1q VLAN, legacy implementations like ISL are not supported.
Create a QinQ interface (VLAN stacking)	*ip link add name ${service interface} link ${physical interface} type vlan proto 802.1ad id ${service tag}*		
	ip link add name ${client interface} link ${service interface} type vlan proto 802.1q id ${client tag}		
	ip link add name eth0.100 link eth0 type vlan proto 802.1ad id 100		Create service tag interface
	ip link add name eth0.100.200 link eth0.100 type vlan proto 802.1q id 200		Create client tag interface
Create pseudo-ethernet (aka macvlan) interface	*ip link add name ${macvlan interface name} link ${parent interface} type macvlan*	*ip link add name peth0 link eth0 type macvlan*	
Create a dummy interface	*ip link add name ${dummy interface name} type dummy*	*ip link add name dummy0 type dummy*	
Create a bridge interface	*ip link add name ${bridge name} type bridge*	*ip link add name br0 type bridge*	
Add an interface to bridge	*ip link set dev ${interface name} master ${bridge name}*	*ip link set dev eth0 master br0*	
Remove interface from bridge	*ip link set dev ${interface name} nomaster*	*ip link set dev eth0 nomaster*	
Create a bonding interface	*ip link add name ${name} type bond*	*ip link add name bond1 type bond*	This is not enough to configure bonding (link aggregation) in any meaningful way. You need to set up bonding parameters according to your situation.

Link management (cont)

Create an intermediate functional block interface	*ip link add ${interface name} type ifb*	*ip link add ifb10 type ifb*	Intermediate functional block devices are used for traffic redirection and mirroring in conjunction with tc.
Create a pair of virtual ethernet devices	*ip link add name ${first device name} type veth peer name ${second device name}*	*ip link add name veth-host type veth peer name veth-guest*	Virtual ethernet devices are created in UP state, no need to bring them up manually after creation.

Note that interface name you set with "name ${name}" parameter of "ip link add" and "ip link set" commands may be arbitrary, and even contain unicode characters. It's better however to stick with ASCII because other programs may not handle unicode correctly. Also it's better to use a consistent convention for link names, and use link aliases to provide human descriptions.

Link group management

Add an interface to a group	*ip link set dev ${interface name} group ${group number}*	*ip link set dev eth0 group 42*	*ip link set dev eth1 group 42*
Remove an interface from a group	*ip link set dev ${interface name} group 0*	*ip link set dev ${interface} group default*	*ip link set dev tun10 group 0*
Assign a symbolic name to a group	*echo "10 customer-vlans" >> /etc/iproute2/group*	Once you configured a group name, number and name can be used interchangeably in ip commands.	*ip link set dev eth0.100 group customer-vlans*
Perform an operation on a group	*ip link set group ${group number} ${operation and arguments}*	*ip link set group 42 down*	*ip link set group uplinks mtu 1200*
View information about links from specific group	*ip link list group 42*	*ip address show group customers*	

Link groups are similar to port ranges found in managed switches. You can add network interfaces to a numbered group and perform operations on all the interfaces from that group at once.

Links not assigned to any group belong to group 0 aka "default".

Tun and Tap devices

Add an tun/tap device useable by root	*ip tuntap add dev ${interface name} mode ${mode}*	*ip tuntap add dev tun0 mode tun*	*ip tuntap add dev tap9 mode tap*
Tap sends and receives raw Ethernet frames.		Tun sends and receives raw IP packets.	
Add an tun/tap device usable by an ordinary user	*ip tuntap add dev ${interface name} mode ${mode} user ${user} group ${group}*	*ip tuntap add dev tun1 mode tun user me group mygroup*	*ip tuntap add dev tun2 mode tun user 1000 group 1001*

Tun and Tap devices (cont)

Add an tun/tap device using an alternate packet format	*ip tuntap add dev ${interface name} mode ${mode} pi*	*ip tuntap add dev tun1 mode tun pi*
Add an tun/tap ignoring flow control	*ip tuntap add dev ${interface name} mode ${mode} one_queue*	*ip tuntap add dev tun1 mode tun one_queue*
Delete tun/tap device	*ip tuntap del dev ${interface name}*	*ip tuntap del dev tun0 name}*

Tun and tap devices allow userspace programs to emulate a network device. When the userspace program opens them they get a file descriptor. Packets routed by the kernel networking stack to the device are read from the file descriptor, data the userspace program writes to the file descriptor are injected as local outgoing packets into the networking stack.

Neighbor (ARP and NDP) tables management

View neighbor tables	*ip neighbor show*	
View neighbors for single interface	*ip neighbor show dev ${interface name}*	*ip neighbor show dev eth0*
Flush table for an interface	*ip neighbor flush dev ${interface name}*	*ip neighbor flush dev eth1*
Add a neighbor table entry	*ip neighbor add ${network address} lladdr ${link layer address} dev ${interface name}*	*ip neighbor add 192.0.2.1 lladdr 22:ce:e0:99:63:6f dev eth0*
Delete a neighbor table entry	*ip neighbor delete ${network address} lladdr ${link layer address} dev ${interface name}*	*ip neighbor delete 192.0.2.1 lladdr 22:ce:e0:99:63:6f dev eth0*

For ladies and gentlemen who prefer UK spelling, this command family supports "neighbour" spelling too.

⟷ Tunnel management

Create an IPIP tunnel	*ip tunnel add ${interface name} mode ipip local ${local endpoint address} remote ${remote endpoint address}*
Create a SIT tunnel	*sudo ip tunnel add ${interface name} mode sit local ${local endpoint address} remote ${remote endpoint address}*
Create an IPIP6 tunnel	*ip -6 tunnel add ${interface name} mode ipip6 local ${local endpoint address} remote ${remote endpoint address}*

⟷ Tunnel management (cont)

Create an IP6IP6 tunnel	*ip -6 tunnel add ${interface name} mode ip6ip6 local ${local endpoint address} remote ${remote endpoint address}*		
Create a gretap (ethernet over GRE) device	*ip link add ${interface name} type gretap local ${local endpoint address} remote ${remote endpoint address}*		
Create a GRE tunnel	*ip tunnel add ${interface name} mode gre local ${local endpoint address} remote ${remote endpoint address}*		
Create multiple GRE tunnels to the same endpoint	*ip tunnel add ${interface name} mode gre local ${local endpoint address} remote ${remote endpoint address} key ${key value}*		
Create a point-to-multipoint GRE tunnel	*ip tunnel add ${interface name} mode gre local ${local endpoint address} key ${key value}*		
Create a GRE tunnel over IPv6	*ip -6 tunnel add name ${interface name} mode ip6gre local ${local endpoint} remote ${remote endpoint}*		
Delete a tunnel	*ip tunnel del ${interface name}*	*ip tunnel del gre1*	
Modify a tunnel	*ip tunnel change ${interface name} ${options}*	*ip tunnel change tun0 remote 203.0.113.89*	*ip tunnel change tun10 key 23456*
View tunnel information	*ip tunnel show*	*ip tunnel show ${interface name}*	*ip tun show tun99*

Linux currently supports IPIP (IPv4 in IPv4), SIT (IPv6 in IPv4), IP6IP6 (IPv6 in IPv6), IPIP6 (IPv4 in IPv6), GRE (virtually anything in anything), and, in very recent versions, VTI (IPv4 in IPsec).

Note that tunnels are created in DOWN state, you need to bring them up.

In this section ${local endpoint address} and ${remote endpoint address} refer to addresses assigned to physical interfaces of endpoint. ${address} refers to the address assigned to tunnel interface.

L2TPv3 pseudowire management

Create an L2TPv3 tunnel over UDP	*ip l2tp add tunnel tunnel_id ${local tunnel numeric identifier} peer_tunnel_id ${remote tunnel numeric identifier} udp_sport ${source port} udp_dport ${destination port} encap udp local ${local endpoint address} remote ${remote endpoint address}*		
	ip l2tp add tunnel tunnel_id 1 peer_tunnel_id 1 udp_sport 5000 udp_dport 5000 encap udp local 192.0.2.1 remote 203.0.113.2		
Create an L2TPv3 tunnel over IP	*ip l2tp add tunnel tunnel_id ${local tunnel numeric identifier} peer_tunnel_id {remote tunnel numeric identifier } encap ip local 192.0.2.1 remote 203.0.113.2*		
Create an L2TPv3 session	*ip l2tp add session tunnel_id ${local tunnel identifier} session_id ${local session numeric identifier} peer_session_id ${remote session numeric identifier}*	*ip l2tp add session tunnel_id 1 session_id 10 peer_session_id 10*	
Delete an L2TPv3 session	*ip l2tp del session tunnel_id ${tunnel identifier} session_id ${session identifier}*	*ip l2tp del session tunnel_id 1 session_id 1*	
Delete an L2TPv3 tunnel	*ip l2tp del tunnel tunnel_id ${tunnel identifier}*	*ip l2tp del tunnel tunnel_id 1*	
View L2TPv3 tunnel information	*ip l2tp show tunnel*	*ip l2tp show tunnel tunnel_id ${tunnel identifier}*	*ip l2tp show tunnel tunnel_id 12*

Cheatography

iproute2 Cheat Sheet

by TME520 (TME520) via cheatography.com/20978/cs/4067/

L2TPv3 pseudowire management (cont)

View L2TPv3 session information	*ip l2tp show session*	*ip l2tp show session session_id ${session identifier} tunnel_id ${tunnel identifier}*	*ip l2tp show session session_id 1 tunnel_id 12*

Compared to other tunneling protocol implementations in Linux, L2TPv3 terminology is somewhat reversed. You create a tunnel, and then bind sessions to it. You can bind multiple sessions with different identifiers to the same tunnel. Virtual network interfaces (by default named l2tpethX) are associated with sessions.

</> Policy-based routing

Create a policy route	*ip route add ${route options} table ${table id or name}*	*ip route add 192.0.2.0/27 via 203.0.113.1 table 10*	*ip route add 2001:db8::/48 dev eth1 table 100*
View policy routes	*ip route show table ${table id or name}*	*ip route show table 100*	*ip route show table test*
General rule syntax	*ip rule add ${options} <lookup ${table id or name}|blackhole|prohibit|unreachable>*		
Create a rule to match a source network	*ip rule add from ${source network} ${action}*	*ip rule add from 192.0.2.0/24 lookup 10*	*ip -6 rule add from 2001:db8::/32 prohibit*
Create a rule to match a destination network	*ip rule add to ${destination network} ${action}*	*ip rule add to 192.0.2.0/24 blackhole*	*ip -6 rule add to 2001:db8::/32 lookup 100*
Create a rule to match a ToS field value	*ip rule add tos ${ToS value} ${action}*	*ip rule add tos 0x10 lookup 110*	
Create a rule to match a firewall mark value	*ip rule add fwmark ${mark} ${action}*	*ip rule add fwmark 0x11 lookup 100*	
Create a rule to match inbound interface	*ip rule add iif ${interface name} ${action}*	*ip rule add iif eth0 lookup 10*	*ip rule add iif lo lookup 20*
Create a rule to match outbound interface	*ip rule add oif ${interface name} ${action}*	*ip rule add oif eth0 lookup 10*	
Set rule priority	*ip rule add ${options} ${action} priority ${value}*	*ip rule add from 192.0.2.0/25 lookup 10 priority 10*	*ip rule add from 192.0.2.0/24 lookup 20 priority 20*
Show all rules	*ip rule show*	*ip -6 rule show*	
Delete a rule	*ip rule del ${options} ${action}*	*ip rule del 192.0.2.0/24 lookup 10*	
Delete all rules	*ip rule flush*	*ip -6 rule flush*	

Policy-based routing (PBR) in Linux is designed the following way: first you create custom routing tables, then you create rules to tell the kernel it should use those tables instead of the default table for specific traffic.

Some tables are predefined: local (table 255), main (table 254), default (table 253).

netconf (sysctl configuration viewing)

View sysctl configuration for all interfaces	*ip netconf show*	
View sysctl configuration for specific interface	*ip netconf show dev ${interface}*	*ip netconf show dev eth0*

Network namespace management

Create a namespace	*ip netns add ${namespace name}*	*ip netns add foo*	
List existing namespaces	*ip netns list*		
Delete a namespace	*ip netns delete ${namespace name}*	*ip netns delete foo*	
Run a process inside a namespace	*ip netns exec ${namespace name} ${command}*	*ip netns exec foo /bin/sh*	
List all processes assigned to a namespace	*ip netns pids ${namespace name}*		The output will be a list of PIDs.
Identify process' primary namespace	*ip netns identify ${pid}*	*ip netns identify 9000*	
Assign network interface to a namespace	*ip link set dev ${interface name} netns ${namespace name}*	*ip link set dev ${interface name} netns ${pid}*	*ip link set dev eth0.100 netns foo*
Connect one namespace to another	Create a pair of veth devices:	*ip link add name veth1 type veth peer name veth2*	
	Move veth2 to namespace foo:	*ip link set dev veth2 netns foo*	
	Bring veth2 and add an address in "foo" namespace:	*ip netns exec foo ip link set dev veth2 up*	
		ip netns exec foo ip address add 10.1.1.1/24 dev veth2	
	Add an address to veth1, which stays in the default namespace:	*ip address add 10.1.1.2/24 dev veth1*	
Monitor network namespace subsystem events	*ip netns monitor*		

Network namespaces are isolated network stack instances within a single machine. They can be used for security domain separation, managing traffic flows between virtual machines and so on.

Every namespace is a complete copy of the networking stack with its own interfaces, addresses, routes etc. You can run processes inside a namespace and bridge namespaces to physical interfaces.

VXLAN management

Create a VXLAN link	*ip link add name ${interface name} type vxlan id <0-16777215> dev ${source interface} group ${multicast address*	*ip link add name vxlan0 type vxlan id 42 dev eth0 group 239.0.0.1*

VXLAN is a layer 2 tunneling protocol that is commonly used in conjunction with virtualization systems such as KVM to connect virtual machines running on different hypervisor nodes to each other and to outside world. The underlying encapsulation protocol for VXLAN is UDP.

Multicast management

View multicast groups	*ip maddress show*	*ip maddress show ${interface name}*	*ip maddress show dev lo*
Add a link-layer multicast address	*ip maddress add ${MAC address} dev ${interface name}*	*ip maddress add 01:00:5e:00:00:ab dev eth0*	
View multicast routes	*ip mroute show*	Multicast routes cannot be added manually, so this command can only show multicast routes installed by a routing daemon.	It supports the same modifiers to unicast route viewing commands (iif, table, from etc.).

Multicast is mostly handled by applications and routing daemons, so there is not much you can and should do manually here. Multicast-related ip commands are mostly useful for debug.

Network event monitoring

Monitor all events	*ip monitor*	
Monitor specific events	*ip monitor ${event type}*	Event type can be: link, address, route, mroute, neigh.
Read a log file produced by rtmon	*ip monitor ${event type} file ${path to the log file}*	iproute2 includes a program called "rtmon" that serves essentially the same purpose, but writes events to a binary log file instead of displaying them. "ip monitor" command allows you to read files created by the program".
	rtmon [-family <inet|inet6>] [<route|link|address|all>] file ${log file path}	rtmon syntax is similar to that of "ip monitor", except event type is limited to link, address, route, and all; and address family is specified in "-family" option.

You can monitor certain network events with iproute2, such as changes in network configuration, routing tables, and ARP/NDP tables.