
Netplan

Netplan team

Nov 29, 2022

CONTENTS

1	Tutorials	1
1.1	Ubuntu 22.04 LTS	1
1.2	Ubuntu 20.04 LTS	1
1.3	Ubuntu 18.04 LTS	1
2	How-to guides	3
2.1	Examples	3
2.2	Use D-Bus config API	15
3	Reference	17
3.1	YAML configuration	17
3.2	API specification	40
3.3	Netplan CLI	40
3.4	Netplan D-Bus	46
4	Explanation	49
4.1	General structure & IDs	49
4.2	Design	50
4.3	FAQs	50
5	Project and community	53

TUTORIALS

Here you can find some external blog posts, describing the usage of Netplan on different systems.

1.1 Ubuntu 22.04 LTS

- <https://vitux.com/how-to-configure-networking-with-netplan-on-ubuntu/>

1.2 Ubuntu 20.04 LTS

- <https://linux-on-z.blogspot.com/p/using-netplan-on-ibm-z.html>
- <https://linuxconfig.org/netplan-network-configuration-tutorial-for-beginners>
- <https://www.serverlab.ca/tutorials/linux/administration-linux/how-to-configure-networking-in-ubuntu-20-04-with-netplan/>

1.3 Ubuntu 18.04 LTS

- <https://www.linux.com/topic/distributions/how-use-netplan-network-configuration-tool-linux/>
- https://linuxhint.com/install_netplan_ubuntu/

HOW-TO GUIDES

2.1 Examples

Below are a collection of example netplan configurations for common scenarios. If you see a scenario missing or have one to contribute, please file a bug against this documentation with the example.

To configure netplan, save configuration files under `/etc/netplan/` with a `.yaml` extension (e.g. `/etc/netplan/config.yaml`), then run `sudo netplan apply`. This command parses and applies the configuration to the system. Configuration written to disk under `/etc/netplan/` will persist between reboots.

Also, see [/examples](#) on GitHub.

2.1.1 Using DHCP and static addressing

To let the interface named `enp3s0` get an address via DHCP, create a YAML file with the following:

```
network:
  version: 2
  renderer: networkd
  ethernets:
    enp3s0:
      dhcp4: true
```

To instead set a static IP address, use the `addresses` key, which takes a list of (IPv4 or IPv6), addresses along with the subnet prefix length (e.g. `/24`). DNS information can be provided as well, and the gateway can be defined via a default route:

```
network:
  version: 2
  renderer: networkd
  ethernets:
    enp3s0:
      addresses:
        - 10.10.10.2/24
      nameservers:
        search: [mydomain, otherdomain]
        addresses: [10.10.10.1, 1.1.1.1]
      routes:
        - to: default
          via: 10.10.10.1
```

2.1.2 Connecting multiple interfaces with DHCP

Many systems now include more than one network interface. Servers will commonly need to connect to multiple networks, and may require that traffic to the Internet goes through a specific interface despite all of them providing a valid gateway.

One can achieve the exact routing desired over DHCP by specifying a metric for the routes retrieved over DHCP, which will ensure some routes are preferred over others. In this example, 'enred' is preferred over 'engreen', as it has a lower route metric:

```
network:
  version: 2
  ethernets:
    enred:
      dhcp4: yes
      dhcp4-overrides:
        route-metric: 100
    engreen:
      dhcp4: yes
      dhcp4-overrides:
        route-metric: 200
```

2.1.3 Connecting to an open wireless network

Netplan easily supports connecting to an open wireless network (one that is not secured by a password), only requiring that the access point is defined:

```
network:
  version: 2
  wifis:
    wl0:
      access-points:
        opennetwork: {}
      dhcp4: yes
```

2.1.4 Connecting to a WPA Personal wireless network

Wireless devices use the 'wifis' key and share the same configuration options with wired ethernet devices. The wireless access point name and password should also be specified:

```
network:
  version: 2
  renderer: networkd
  wifis:
    wlp2s0b1:
      dhcp4: no
      dhcp6: no
      addresses: [192.168.0.21/24]
      nameservers:
        addresses: [192.168.0.1, 8.8.8.8]
      access-points:
```

(continues on next page)

(continued from previous page)

```

    "network_ssid_name":
      password: "*****"
  routes:
    - to: default
      via: 192.168.0.1

```

2.1.5 Connecting to WPA Enterprise wireless networks

It is also common to find wireless networks secured using WPA or WPA2 Enterprise, which requires additional authentication parameters.

For example, if the network is secured using WPA-EAP and TTLS:

```

network:
  version: 2
  wifis:
    wl0:
      access-points:
        workplace:
          auth:
            key-management: eap
            method: ttls
            anonymous-identity: "@internal.example.com"
            identity: "joe@internal.example.com"
            password: "v3ryS3kr1t"
      dhcp4: yes

```

Or, if the network is secured using WPA-EAP and TLS:

```

network:
  version: 2
  wifis:
    wl0:
      access-points:
        university:
          auth:
            key-management: eap
            method: tls
            anonymous-identity: "@cust.example.com"
            identity: "cert-joe@cust.example.com"
            ca-certificate: /etc/ssl/cust-cacrt.pem
            client-certificate: /etc/ssl/cust-crt.pem
            client-key: /etc/ssl/cust-key.pem
            client-key-password: "d3cryptPr1v4t3K3y"
      dhcp4: yes

```

Many different modes of encryption are supported. See the *Netplan reference* page.

2.1.6 Using multiple addresses on a single interface

The addresses key can take a list of addresses to assign to an interface:

```
network:
  version: 2
  renderer: networkd
  ethernets:
    enp3s0:
      addresses:
        - 10.100.1.37/24
        - 10.100.1.38/24:
            label: "enp3s0:0"
        - 10.100.1.39/24:
            label: "enp3s0:some-label"
      routes:
        - to: default
          via: 10.100.1.1
```

2.1.7 Using multiple addresses with multiple gateways

Similar to the example above, interfaces with multiple addresses can be configured with multiple gateways, and static DNS nameservers (Google DNS for this example):

```
network:
  version: 2
  renderer: networkd
  ethernets:
    enp3s0:
      addresses:
        - 10.0.0.10/24
        - 11.0.0.11/24
      nameservers:
        addresses:
          - 8.8.8.8
          - 8.8.4.4
      routes:
        - to: default
          via: 10.0.0.1
          metric: 200
        - to: default
          via: 11.0.0.1
          metric: 300
```

We configure individual routes to default (or 0.0.0.0/0) using the address of the gateway for the subnet. The `metric` value should be adjusted so the routing happens as expected.

DHCP can be used to receive one of the IP addresses for the interface. In this case, the default route for that address will be automatically configured with a `metric` value of 100.

2.1.8 Using Network Manager as a renderer

Netplan supports both networkd and Network Manager as backends. You can specify which network backend should be used to configure particular devices by using the `renderer` key. You can also delegate all configuration of the network to Network Manager itself by specifying only the `renderer` key:

```
network:
  version: 2
  renderer: NetworkManager
```

2.1.9 Configuring interface bonding

Bonding is configured by declaring a bond interface with a list of physical interfaces and a bonding mode. Below is an example of an active-backup bond that uses DHCP to obtain an address:

```
network:
  version: 2
  renderer: networkd
  bonds:
    bond0:
      dhcp4: yes
      interfaces:
        - enp3s0
        - enp4s0
      parameters:
        mode: active-backup
        primary: enp3s0
```

Below is an example of a system acting as a router with various bonded interfaces and different types. Note the ‘optional: true’ key declarations that allow booting to occur without waiting for those interfaces to activate fully.

```
network:
  version: 2
  renderer: networkd
  ethernets:
    enp1s0:
      dhcp4: no
    enp2s0:
      dhcp4: no
    enp3s0:
      dhcp4: no
      optional: true
    enp4s0:
      dhcp4: no
      optional: true
    enp5s0:
      dhcp4: no
      optional: true
    enp6s0:
      dhcp4: no
      optional: true
  bonds:
```

(continues on next page)

(continued from previous page)

```
bond-lan:
  interfaces: [enp2s0, enp3s0]
  addresses: [192.168.93.2/24]
  parameters:
    mode: 802.3ad
    mii-monitor-interval: 1
bond-wan:
  interfaces: [enp1s0, enp4s0]
  addresses: [192.168.1.252/24]
  nameservers:
    search: [local]
    addresses: [8.8.8.8, 8.8.4.4]
  parameters:
    mode: active-backup
    mii-monitor-interval: 1
    gratuitous-arp: 5
  routes:
    - to: default
      via: 192.168.1.1
bond-comntrack:
  interfaces: [enp5s0, enp6s0]
  addresses: [192.168.254.2/24]
  parameters:
    mode: balance-rr
    mii-monitor-interval: 1
```

2.1.10 Configuring network bridges

To create a very simple bridge consisting of a single device that uses DHCP, write:

```
network:
  version: 2
  renderer: networkd
  ethernets:
    enp3s0:
      dhcp4: no
  bridges:
    br0:
      dhcp4: yes
      interfaces:
        - enp3s0
```

A more complex example, to get libvirtd to use a specific bridge with a tagged vlan, while continuing to provide an untagged interface as well would involve:

```
network:
  version: 2
  renderer: networkd
  ethernets:
    enp0s25:
      dhcp4: true
```

(continues on next page)

(continued from previous page)

```

bridges:
  br0:
    addresses: [ 10.3.99.25/24 ]
    interfaces: [ vlan15 ]
vlans:
  vlan15:
    accept-ra: no
    id: 15
    link: enp0s25

```

Then libvirtd would be configured to use this bridge by adding the following content to a new XML file under `/etc/libvirt/qemu/networks/`. The name of the bridge in the `<bridge>` tag as well as in `<name>` need to match the name of the bridge device configured using netplan:

```

<network>
  <name>br0</name>
  <bridge name='br0' />
  <forward mode="bridge"/>
</network>

```

2.1.11 Attaching VLANs to network interfaces

To configure multiple VLANs with renamed interfaces:

```

network:
  version: 2
  renderer: networkd
  ethernets:
    mainif:
      match:
        macaddress: "de:ad:be:ef:ca:fe"
      set-name: mainif
      addresses: [ "10.3.0.5/23" ]
      nameservers:
        addresses: [ "8.8.8.8", "8.8.4.4" ]
        search: [ example.com ]
      routes:
        - to: default
          via: 10.3.0.1
  vlans:
    vlan15:
      id: 15
      link: mainif
      addresses: [ "10.3.99.5/24" ]
    vlan10:
      id: 10
      link: mainif
      addresses: [ "10.3.98.5/24" ]
      nameservers:
        addresses: [ "127.0.0.1" ]
        search: [ domain1.example.com, domain2.example.com ]

```

2.1.12 Reaching a directly connected gateway

This allows setting up a default route, or any route, using the “on-link” keyword where the gateway is an IP address that is directly connected to the network even if the address does not match the subnet configured on the interface.

```
network:
  version: 2
  renderer: networkd
  ethernets:
    ens3:
      addresses: [ "10.10.10.1/24" ]
      routes:
        - to: default # or 0.0.0.0/0
          via: 9.9.9.9
          on-link: true
```

For IPv6 the config would be very similar, with the notable difference being an additional scope: link host route to the router’s address required:

```
network:
  version: 2
  renderer: networkd
  ethernets:
    ens3:
      addresses: [ "2001:cafe:face:beef::dead:dead/64" ]
      routes:
        - to: "2001:cafe:face::1/128"
          scope: link
        - to: default # or "::/0"
          via: "2001:cafe:face::1"
          on-link: true
```

2.1.13 Configuring source routing

Route tables can be added to particular interfaces to allow routing between two networks:

In the example below, ens3 is on the 192.168.3.0/24 network and ens5 is on the 192.168.5.0/24 network. This enables clients on either network to connect to the other and allow the response to come from the correct interface.

Furthermore, the default route is still assigned to ens5 allowing any other traffic to go through it.

```
network:
  version: 2
  renderer: networkd
  ethernets:
    ens3:
      addresses:
        - 192.168.3.30/24
      dhcp4: no
      routes:
        - to: 192.168.3.0/24
          via: 192.168.3.1
          table: 101
```

(continues on next page)

(continued from previous page)

```
routing-policy:
- from: 192.168.3.0/24
  table: 101
ens5:
addresses:
- 192.168.5.24/24
dhcp4: no
routes:
- to: default
  via: 192.168.5.1
- to: 192.168.5.0/24
  via: 192.168.5.1
  table: 102
routing-policy:
- from: 192.168.5.0/24
  table: 102
```

2.1.14 Configuring a loopback interface

Networkd does not allow creating new loopback devices, but a user can add new addresses to the standard loopback interface, lo, in order to have it considered a valid address on the machine as well as for custom routing:

```
network:
version: 2
renderer: networkd
ethernets:
lo:
addresses: [ "127.0.0.1/8", "::1/128", "7.7.7.7/32" ]
```

2.1.15 Integration with a Windows DHCP Server

For networks where DHCP is provided by a Windows Server using the dhcp-identifier key allows for interoperability:

```
network:
version: 2
ethernets:
enp3s0:
dhcp4: yes
dhcp-identifier: mac
```

2.1.16 Connecting an IP tunnel

Tunnels allow an administrator to extend networks across the Internet by configuring two endpoints that will connect a special tunnel interface and do the routing required. Netplan supports SIT, GRE, IP-in-IP (ipip, ipip6, ip6ip6), IP6GRE, VTI and VTI6 tunnels.

A common use of tunnels is to enable IPv6 connectivity on networks that only support IPv4. The example below show how such a tunnel might be configured.

Here, 1.1.1.1 is the client's own IP address; 2.2.2.2 is the remote server's IPv4 address, "2001:dead:beef::2/64" is the client's IPv6 address as defined by the tunnel, and "2001:dead:beef::1" is the remote server's IPv6 address.

Finally, "2001:cafe:face::1/64" is an address for the client within the routed IPv6 prefix:

```
network:
  version: 2
  ethernets:
    eth0:
      addresses:
        - 1.1.1.1/24
        - "2001:cafe:face::1/64"
      routes:
        - to: default
          via: 1.1.1.254
  tunnels:
    he-ipv6:
      mode: sit
      remote: 2.2.2.2
      local: 1.1.1.1
      addresses:
        - "2001:dead:beef::2/64"
      routes:
        - to: default
          via: "2001:dead:beef::1"
```

2.1.17 Configuring SR-IOV Virtual Functions

For SR-IOV network cards, it is possible to dynamically allocate Virtual Function interfaces for every configured Physical Function. In netplan, a VF is defined by having a link: property pointing to the parent PF.

```
network:
  version: 2
  ethernets:
    eno1:
      mtu: 9000
    enp1s16f1:
      link: eno1
      addresses : [ "10.15.98.25/24" ]
    vf1:
      match:
        name: enp1s16f[2-3]
      link: eno1
      addresses : [ "10.15.99.25/24" ]
```

2.1.18 Complex example

This is a complex example which shows most available features

```

network:
  version: 2
  # if specified, can only realistically have that value, as networkd cannot
  # render wifi/3G.
  renderer: NetworkManager
  vrfs:
    mgmt-vrf:
      table: 10
      interfaces:
        - id1
      routes:
        - to: default
          via: 192.168.24.254
          metric: 100
  ethernets:
    lo:
      addresses:
        - 172.16.20.20/32
      link-local: []
      # opaque ID for physical interfaces, only referred to by other stanzas
      id0:
        match:
          macaddress: 00:11:22:33:44:55
        wakeonlan: true
        dhcp4: true
        addresses:
          - 192.168.14.2/24
          - 192.168.14.3/24
          - "2001:1::1/64"
        nameservers:
          search: [foo.local, bar.local]
          addresses: [8.8.8.8]
        routes:
          - to: default
            via: 192.168.14.1
          - to: default
            via: "2001:1::2"
          - to: 0.0.0.0/0
            via: 11.0.0.1
            table: 70
            on-link: true
            metric: 3
        routing-policy:
          - to: 10.0.0.0/8
            from: 192.168.14.2/24
            table: 70
            priority: 100
          - to: 20.0.0.0/8
            from: 192.168.14.3/24

```

(continues on next page)

```
    table: 70
    priority: 50
    # only networkd can render on-link routes and routing policies
    renderer: networkd
id1:
  match:
    macaddress: 00:11:22:33:44:56
  wakeonlan: true
  dhcp4: true
  addresses:
    - 192.168.24.2/24
lom:
  match:
    driver: ixgbe
    # you are responsible for setting tight enough match rules
    # that only match one device if you use set-name
  set-name: lom1
  dhcp6: true
switchports:
  # all cards on second PCI bus unconfigured by
  # themselves, will be added to br0 below
  match:
    name: enp2*
  mtu: 1280
wifis:
  all-wlans:
    # useful on a system where you know there is
    # only ever going to be one device
    match: {}
    access-points:
      "Joe's home":
        # mode defaults to "infrastructure" (client)
        password: "s3kr1t"
    # this creates an AP on wlp1s0 using hostapd
    # no match rules, thus the ID is the interface name
  wlp1s0:
    access-points:
      "guest":
        mode: ap
        # no WPA config implies default of open
bridges:
  # the key name is the name for virtual (created) interfaces
  # no match: and set-name: allowed
  br0:
    # IDs of the components; switchports expands into multiple interfaces
    interfaces: [wlp1s0, switchports]
    dhcp4: true
  br20:
    interfaces: [vxlan20]
tunnels:
  vxlan20:
    mode: vxlan
```

(continues on next page)

(continued from previous page)

```

link: lo
id: 20
mtu: 8950
accept-ra: no
neigh-suppress: true
link-local: []
mac-learning: false
port: 4789
local: 172.16.20.20

```

2.2 Use D-Bus config API

See also:

- [Netplan D-Bus reference](#)
- [busctl reference](#)

Copy the current state from `{etc,run,lib}/netplan/*.yaml` by creating a new config object

```

$ busctl call io.netplan.Netplan /io/netplan/Netplan io.netplan.Netplan Config
o "/io/netplan/Netplan/config/ULJIU0"

```

Read the merged YAML configuration

```

$ busctl call io.netplan.Netplan /io/netplan/Netplan/config/ULJIU0 io.netplan.Netplan.
↳Config Get
s "network:\n ethernets:\n   eth0:\n     dhcp4: true\n   renderer: networkd\n
↳version: 2\n"

```

Write a new config snippet into `70-snapd.yaml`

```

$ busctl call io.netplan.Netplan /io/netplan/Netplan/config/ULJIU0 io.netplan.Netplan.
↳Config Set ss "ethernets.eth0={dhcp4: false, dhcp6: true}" "70-snapd"
b true

```

Check the newly written configuration

```

$ busctl call io.netplan.Netplan /io/netplan/Netplan/config/ULJIU0 io.netplan.Netplan.
↳Config Get
s "network:\n ethernets:\n   eth0:\n     dhcp4: false\n     dhcp6: true\n   renderer:
↳networkd\n version: 2\n"

```

Try to apply the current config object's state

```

$ busctl call io.netplan.Netplan /io/netplan/Netplan/config/ULJIU0 io.netplan.Netplan.
↳Config Try u 20
b true

```

Accept the `Try()` state within the 20 seconds timeout, if not it will be auto-rejected

```
$ busctl call io.netplan.Netplan /io/netplan/Netplan/config/ULJIU0 io.netplan.Netplan.  
↳Config Apply  
b true  
  
[SIGNAL] io.netplan.Netplan /io/netplan/Netplan/config/ULJIU0 io.netplan.Netplan.Config_␣  
↳Changed() is triggered  
[OBJECT] io.netplan.Netplan /io/netplan/Netplan/config/ULJIU0 is removed from the bus
```

Create a new config object and get the merged YAML config

```
$ busctl call io.netplan.Netplan /io/netplan/Netplan io.netplan.Netplan Config  
o "/io/netplan/Netplan/config/KC0IU0  
$ busctl call io.netplan.Netplan /io/netplan/Netplan/config/KC0IU0 io.netplan.Netplan.  
↳Config Get  
s "network:\n ethernets:\n   eth0:\n     dhcp4: false\n     dhcp6: true\n   renderer:␣  
↳networkd\n   version: 2\n"
```

Reject that config object again

```
$ busctl call io.netplan.Netplan /io/netplan/Netplan/config/KC0IU0 io.netplan.Netplan.  
↳Config Cancel  
b true  
  
[SIGNAL] io.netplan.Netplan /io/netplan/Netplan/config/KC0IU0 io.netplan.Netplan.Config_␣  
↳Changed() is triggered  
[OBJECT] io.netplan.Netplan /io/netplan/Netplan/config/KC0IU0 is removed from the bus
```

3.1 YAML configuration

Netplan's configuration files use the [YAML](#) format. All `/lib,etc,run/netplan/*.yaml` are considered.

The top-level node in a netplan configuration file is a `network:` mapping that contains `version: 2` (the YAML currently being used by curtin, MaaS, etc. is version 1), and then device definitions grouped by their type, such as `ethernets:`, `modems:`, `wifis:`, or `bridges:`. These are the types that our renderer can understand and are supported by our backends.

3.1.1 YAML configuration

Properties for physical device types

Note: Some options will not work reliably for devices matched by name only and rendered by networkd, due to interactions with device renaming in udev. Match devices by MAC when setting options like: `wakeonlan` or `*-offload`.

- **match** (mapping)

This selects a subset of available physical devices by various hardware properties. The following configuration will then apply to all matching devices, as soon as they appear. *All* specified properties must match.

- **name** (scalar)

Current interface name. Globs are supported, and the primary use case for matching on names, as selecting one fixed name can be more easily achieved with having no `match:` at all and just using the ID (see above). (NetworkManager: as of v1.14.0)

- **macaddress** (scalar)

Device's 6-byte MAC address in the form "XX:XX:XX:XX:XX:XX" or 20 bytes for InfiniBand devices (IPoIB). Globs are not allowed.

- **driver** (scalar or sequence of scalars) – sequence since **0.104**

Kernel driver name, corresponding to the DRIVER udev property. A sequence of globs is supported, any of which must match. Matching on driver is *only* supported with networkd.

Examples:

- All cards on second PCI bus:

```
match:  
  name: enp2*
```

- Fixed MAC address:

```
match:
  macaddress: 11:22:33:AA:BB:FF
```

- First card of driver ixgbe:

```
match:
  driver: ixgbe
  name: en*s0
```

- First card with a driver matching bcmgenet or smsc*:

```
match:
  driver: ["bcmgenet", "smsc*"]
  name: en*
```

- **set-name** (scalar)

When matching on unique properties such as path or MAC, or with additional assumptions such as “there will only ever be one wifi device”, match rules can be written so that they only match one device. Then this property can be used to give that device a more specific/desirable/nicer name than the default from udev’s ifnames. Any additional device that satisfies the match rules will then fail to get renamed and keep the original kernel name (and dmesg will show an error).

- **wakeonlan** (bool)

Enable wake on LAN. Off by default.

- **emit-lldp** (bool) – since **0.99**

(networkd backend only) Whether to emit LLDP packets. Off by default.

- **receive-checksum-offload** (bool) – since **0.104**

(networkd backend only) If set to true (false), the hardware offload for checksumming of ingress network packets is enabled (disabled). When unset, the kernel’s default will be used.

- **transmit-checksum-offload** (bool) – since **0.104**

(networkd backend only) If set to true (false), the hardware offload for checksumming of egress network packets is enabled (disabled). When unset, the kernel’s default will be used.

- **tcp-segmentation-offload** (bool) – since **0.104**

(networkd backend only) If set to true (false), the TCP Segmentation Offload (TSO) is enabled (disabled). When unset, the kernel’s default will be used.

- **tcp6-segmentation-offload** (bool) – since **0.104**

(networkd backend only) If set to true (false), the TCP6 Segmentation Offload (tx-tcp6-segmentation) is enabled (disabled). When unset, the kernel’s default will be used.

- **generic-segmentation-offload** (bool) – since **0.104**

(networkd backend only) If set to true (false), the Generic Segmentation Offload (GSO) is enabled (disabled). When unset, the kernel’s default will be used.

- **generic-receive-offload** (bool) – since **0.104**

(networkd backend only) If set to true (false), the Generic Receive Offload (GRO) is enabled (disabled). When unset, the kernel’s default will be used.

- **large-receive-offload** (bool) – since **0.104**

(networkd backend only) If set to true (false), the Large Receive Offload (LRO) is enabled (disabled). When unset, the kernel's default will be used.

- **openvswitch** (mapping) – since **0.100**

This provides additional configuration for the openvswitch network device. If Open vSwitch is not available on the system, netplan treats the presence of `openvswitch` configuration as an error.

Any supported network device that is declared with the `openvswitch` mapping (or any bond/bridge that includes an interface with an openvswitch configuration) will be created in openvswitch instead of the defined renderer. In the case of a `vlan` definition declared the same way, netplan will create a fake VLAN bridge in openvswitch with the requested `vlan` properties.

- **external-ids** (mapping) – since **0.100**

Passed-through directly to Open vSwitch

- **other-config** (mapping) – since **0.100**

Passed-through directly to Open vSwitch

- **lACP** (scalar) – since **0.100**

Valid for bond interfaces. Accepts `active`, `passive` or `off` (the default).

- **fail-mode** (scalar) – since **0.100**

Valid for bridge interfaces. Accepts `secure` or `standalone` (the default).

- **mcast-snooping** (bool) – since **0.100**

Valid for bridge interfaces. False by default.

- **protocols** (sequence of scalars) – since **0.100**

Valid for bridge interfaces or the network section. List of protocols to be used when negotiating a connection with the controller. Accepts `OpenFlow10`, `OpenFlow11`, `OpenFlow12`, `OpenFlow13`, `OpenFlow14`, `OpenFlow15` and `OpenFlow16`.

- **rstp** (bool) – since **0.100**

Valid for bridge interfaces. False by default.

- **controller** (mapping) – since **0.100**

Valid for bridge interfaces. Specify an external OpenFlow controller.

- * **addresses** (sequence of scalars)

Set the list of addresses to use for the controller targets. The syntax of these addresses is as defined in `ovs-vsctl(8)`. Example: `addresses: [tcp:127.0.0.1:6653, "ssl:[fe80::1234%eth0]:6653"]`

- * **connection-mode** (scalar)

Set the connection mode for the controller. Supported options are `in-band` and `out-of-band`. The default is `in-band`.

- **ports** (sequence of sequence of scalars) – since **0.100**

Open vSwitch patch ports. Each port is declared as a pair of names which can be referenced as interfaces in dependent virtual devices (bonds, bridges).

Example:

```

openvswitch:
  ports:
    - [patch0-1, patch1-0]

```

– **ssl** (mapping) – since **0.100**

Valid for global `openvswitch` settings. Options for configuring SSL server endpoint for the switch.

* **ca-cert** (scalar)

Path to a file containing the CA certificate to be used.

* **certificate** (scalar)

Path to a file containing the server certificate.

* **private-key** (scalar)

Path to a file containing the private key for the server.

Properties for all device types

• **renderer** (scalar)

Use the given networking backend for this definition. Currently supported are `networkd` and `NetworkManager`. This property can be specified globally in `network:`, for a device type (in e. g. `ethernets:`) or for a particular device definition. Default is `networkd`.

(Since 0.99) The `renderer` property has one additional acceptable value for `vlan` objects (i. e. defined in `vlans:`): `sriov`. If a `vlan` is defined with the `sriov` `renderer` for an SR-IOV Virtual Function interface, this causes netplan to set up a hardware VLAN filter for it. There can be only one defined per VF.

• **dhcp4** (bool)

Enable DHCP for IPv4. Off by default.

• **dhcp6** (bool)

Enable DHCP for IPv6. Off by default. This covers both stateless DHCP - where the DHCP server supplies information like DNS nameservers but not the IP address - and stateful DHCP, where the server provides both the address and the other information.

If you are in an IPv6-only environment with completely stateless auto-configuration (SLAAC with RDNSS), this option can be set to cause the interface to be brought up. (Setting `accept-ra` alone is not sufficient.) Auto-configuration will still honor the contents of the router advertisement and only use DHCP if requested in the RA.

Note that `rdnssd(8)` is required to use RDNSS with `networkd`. No extra software is required for `NetworkManager`.

• **ipv6-mtu** (scalar) – since **0.98**

Set the IPv6 MTU (only supported with `networkd` backend). Note that needing to set this is an unusual requirement.

Requires feature: `ipv6-mtu`

• **ipv6-privacy** (bool)

Enable IPv6 Privacy Extensions (RFC 4941) for the specified interface, and prefer temporary addresses. Defaults to false - no privacy extensions. There is currently no way to have a private address but prefer the public address.

- **link-local** (sequence of scalars)

Configure the link-local addresses to bring up. Valid options are ‘ipv4’ and ‘ipv6’, which respectively allow enabling IPv4 and IPv6 link local addressing. If this field is not defined, the default is to enable only IPv6 link-local addresses. If the field is defined but configured as an empty set, IPv6 link-local addresses are disabled as well as IPv4 link- local addresses.

This feature enables or disables link-local addresses for a protocol, but the actual implementation differs per backend. On networkd, this directly changes the behavior and may add an extra address on an interface. When using the NetworkManager backend, enabling link-local has no effect if the interface also has DHCP enabled.

Examples:

- Enable only IPv4 link-local: `link-local: [ipv4]`
- Enable all link-local addresses: `link-local: [ipv4, ipv6]`
- Disable all link-local addresses: `link-local: []`

- **ignore-carrier** (bool) – since **0.104**

(networkd backend only) Allow the specified interface to be configured even if it has no carrier.

- **critical** (bool)

Designate the connection as “critical to the system”, meaning that special care will be taken by to not release the assigned IP when the daemon is restarted. (not recognized by NetworkManager)

- **dhcp-identifier** (scalar)

(networkd backend only) Sets the source of DHCPv4 client identifier. If `mac` is specified, the MAC address of the link is used. If this option is omitted, or if `duid` is specified, networkd will generate an RFC4361-compliant client identifier for the interface by combining the link’s IAID and DUID.

- **dhcp4-overrides** (mapping)

(networkd backend only) Overrides default DHCP behavior; see the [DHCP Overrides](#) section below.

- **dhcp6-overrides** (mapping)

(networkd backend only) Overrides default DHCP behavior; see the [DHCP Overrides](#) section below.

- **accept-ra** (bool)

Accept Router Advertisement that would have the kernel configure IPv6 by itself. When enabled, accept Router Advertisements. When disabled, do not respond to Router Advertisements. If unset use the host kernel default setting.

- **addresses** (sequence of scalars and mappings)

Add static addresses to the interface in addition to the ones received through DHCP or RA. Each sequence entry is in CIDR notation, i. e. of the form `addr/prefixlen`. `addr` is an IPv4 or IPv6 address as recognized by `inet_pton(3)` and `prefixlen` the number of bits of the subnet.

For virtual devices (bridges, bonds, vlan) if there is no address configured and DHCP is disabled, the interface may still be brought online, but will not be addressable from the network.

In addition to the addresses themselves one can specify configuration parameters as mappings. Current supported options are:

- **lifetime** (scalar) – since **0.100**

Default: `forever`. This can be `forever` or `0` and corresponds to the `PreferredLifetime` option in `systemd-networkd`'s Address section. Currently supported on the `networkd` backend only.

- **label** (scalar) – since **0.100**

An IP address label, equivalent to the `ip address label` command. Currently supported on the `networkd` backend only.

Examples:

- Simple: `addresses: [192.168.14.2/24, "2001:1::1/64"]`
- Advanced:

```
ethernets:
  eth0:
    addresses:
      - "10.0.0.15/24":
        lifetime: 0
        label: "maas"
      - "2001:1::1/64"
```

- **ipv6-address-generation** (scalar) – since **0.99**

Configure method for creating the address for use with RFC4862 IPv6 Stateless Address Auto-configuration (only supported with `NetworkManager` backend). Possible values are `eui64` or `stable-privacy`.

- **ipv6-address-token** (scalar) – since **0.100**

Define an IPv6 address token for creating a static interface identifier for IPv6 Stateless Address Auto-configuration. This is mutually exclusive with `ipv6-address-generation`.

- **gateway4, gateway6** (scalar)

Deprecated, see `Default routes`. Set default gateway for IPv4/6, for manual address configuration. This requires setting `addresses` too. Gateway IPs must be in a form recognized by `inet_pton(3)`. There should only be a single gateway per IP address family set in your global config, to make it unambiguous. If you need multiple default routes, please define them via `routing-policy`.

Examples

- IPv4: `gateway4: 172.16.0.1`
- IPv6: `gateway6: "2001:4::1"`

- **nameservers** (mapping)

Set DNS servers and search domains, for manual address configuration. There are two supported fields: `addresses`: is a list of IPv4 or IPv6 addresses similar to `gateway*`, and `search`: is a list of search domains.

Example:

```
ethernets:
  id0:
    [...]
  nameservers:
    search: [lab, home]
    addresses: [8.8.8.8, "FEDC::1"]
```

- **macaddress** (scalar)

Set the device’s MAC address. The MAC address must be in the form “XX:XX:XX:XX:XX:XX”.

Note: This will not work reliably for devices matched by name only and rendered by networkd, due to interactions with device renaming in udev. Match devices by MAC when setting MAC addresses.

Example:

```
ethernets:
  id0:
    match:
      macaddress: 52:54:00:6b:3c:58
    [...]
    macaddress: 52:54:00:6b:3c:59
```

- **mtu** (scalar)

Set the Maximum Transmission Unit for the interface. The default is 1500. Valid values depend on your network interface.

Note: This will not work reliably for devices matched by name only and rendered by networkd, due to interactions with device renaming in udev. Match devices by MAC when setting MTU.

- **optional** (bool)

An optional device is not required for booting. Normally, networkd will wait some time for device to become configured before proceeding with booting. However, if a device is marked as optional, networkd will not wait for it. This is *only* supported by networkd, and the default is false.

Example:

```
ethernets:
  eth7:
    # this is plugged into a test network that is often
    # down - don't wait for it to come up during boot.
    dhcp4: true
    optional: true
```

- **optional-addresses** (sequence of scalars)

Specify types of addresses that are not required for a device to be considered online. This changes the behavior of backends at boot time to avoid waiting for addresses that are marked optional, and thus consider the interface as “usable” sooner. This does not disable these addresses, which will be brought up anyway.

Example:

```
ethernets:
  eth7:
    dhcp4: true
    dhcp6: true
    optional-addresses: [ ipv4-ll, dhcp6 ]
```

- **activation-mode** (scalar) – since **0.103**

Allows specifying the management policy of the selected interface. By default, netplan brings up any configured interface if possible. Using the `activation-mode` setting users can override that behavior by either specifying `manual`, to hand over control over the interface state to the administrator or (for networkd backend *only*) `off` to force the link in a down state at all times. Any interface with

activation-mode defined is implicitly considered optional. Supported officially as of networkd v248+.

Example:

```
ethernets:
  eth1:
    # this interface will not be put into an UP state automatically
    dhcp4: true
    activation-mode: manual
```

- **routes** (sequence of mappings)
 - Configure static routing for the device; see the [Routing](#) section below.
- **routing-policy** (sequence of mappings)
 - Configure policy routing for the device; see the [Routing](#) section below.
- **neigh-suppress** (scalar) – since **0.105**
 - Takes a boolean. Configures whether ARP and ND neighbor suppression is enabled for this port. When unset, the kernel’s default will be used.

DHCP Overrides

Several DHCP behavior overrides are available. Most currently only have any effect when using the `networkd` backend, with the exception of `use-routes` and `route-metric`.

Overrides only have an effect if the corresponding `dhcp4` or `dhcp6` is set to `true`.

If both `dhcp4` and `dhcp6` are `true`, the `networkd` backend requires that `dhcp4-overrides` and `dhcp6-overrides` contain the same keys and values. If the values do not match, an error will be shown and the network configuration will not be applied.

When using the `NetworkManager` backend, different values may be specified for `dhcp4-overrides` and `dhcp6-overrides`, and will be applied to the DHCP client processes as specified in the netplan YAML.

- **dhcp4-overrides, dhcp6-overrides** (mapping)
 - The `dhcp4-overrides` and `dhcp6-overrides` mappings override the default DHCP behavior.
 - **use-dns** (bool)
 - Default: `true`. When `true`, the DNS servers received from the DHCP server will be used and take precedence over any statically configured ones. Currently only has an effect on the `networkd` backend.
 - **use-ntp** (bool)
 - Default: `true`. When `true`, the NTP servers received from the DHCP server will be used by `systemd-timesyncd` and take precedence over any statically configured ones. Currently only has an effect on the `networkd` backend.
 - **send-hostname** (bool)
 - Default: `true`. When `true`, the machine’s hostname will be sent to the DHCP server. Currently only has an effect on the `networkd` backend.
 - **use-hostname** (bool)
 - Default: `true`. When `true`, the hostname received from the DHCP server will be set as the transient hostname of the system. Currently only has an effect on the `networkd` backend.

- **use-mtu** (bool)

Default: `true`. When `true`, the MTU received from the DHCP server will be set as the MTU of the network interface. When `false`, the MTU advertised by the DHCP server will be ignored. Currently only has an effect on the `networkd` backend.
- **hostname** (scalar)

Use this value for the hostname which is sent to the DHCP server, instead of machine’s hostname. Currently only has an effect on the `networkd` backend.
- **use-routes** (bool)

Default: `true`. When `true`, the routes received from the DHCP server will be installed in the routing table normally. When set to `false`, routes from the DHCP server will be ignored: in this case, the user is responsible for adding static routes if necessary for correct network operation. This allows users to avoid installing a default gateway for interfaces configured via DHCP. Available for both the `networkd` and `NetworkManager` backends.
- **route-metric** (scalar)

Use this value for default metric for automatically-added routes. Use this to prioritize routes for devices by setting a lower metric on a preferred interface. Available for both the `networkd` and `NetworkManager` backends.
- **use-domains** (scalar) – since **0.98**

Takes a boolean, or the special value “route”. When true, the domain name received from the DHCP server will be used as DNS search domain over this link, similar to the effect of the `Domains=` setting. If set to “route”, the domain name received from the DHCP server will be used for routing DNS queries only, but not for searching, similar to the effect of the `Domains=` setting when the argument is prefixed with “~”.

Requires feature: `dhcp-use-domains`

Routing

Complex routing is possible with netplan. Standard static routes as well as policy routing using routing tables are supported via the `networkd` backend.

These options are available for all types of interfaces.

Default routes

The most common need for routing concerns the definition of default routes to reach the wider Internet. Those default routes can only be defined once per IP family and routing table. A typical example would look like the following:

```
eth0:
  [...]
  routes:
    - to: default # could be 0.0.0.0/0 optionally
      via: 10.0.0.1
      metric: 100
      on-link: true
    - to: default # could be ::/0 optionally
      via: cf02:de:ad:be:ef::2
eth1:
```

(continues on next page)

```
[...]
routes:
- to: default
  via: 172.134.67.1
  metric: 100
  on-link: true
  # Not on the main routing table,
  # does not conflict with the eth0 default route
  table: 76
```

- **routes** (mapping)

The `routes` block defines standard static routes for an interface. At least `to` must be specified. If type is `local` or `nat` a default scope of `host` is assumed. If type is `unicast` and no gateway (`via`) is given or type is `broadcast`, `multicast` or `anycast` a default scope of `link` is assumed. Otherwise, a global scope is the default setting.

For `from`, `to`, and `via`, both IPv4 and IPv6 addresses are recognized, and must be in the form `addr/prefixlen` or `addr`.

- **from** (scalar)

Set a source IP address for traffic going through the route. (NetworkManager: as of v1.8.0)

- **to** (scalar)

Destination address for the route.

- **via** (scalar)

Address to the gateway to use for this route.

- **on-link** (bool)

When set to “true”, specifies that the route is directly connected to the interface. (NetworkManager: as of v1.12.0 for IPv4 and v1.18.0 for IPv6)

- **metric** (scalar)

The relative priority of the route. Must be a positive integer value.

- **type** (scalar)

The type of route. Valid options are “unicast” (default), “anycast”, “blackhole”, “broadcast”, “local”, “multicast”, “nat”, “prohibit”, “throw”, “unreachable” or “xresolve”.

- **scope** (scalar)

The route scope, how wide-ranging it is to the network. Possible values are “global”, “link”, or “host”.

- **table** (scalar)

The table number to use for the route. In some scenarios, it may be useful to set routes in a separate routing table. It may also be used to refer to routing policy rules which also accept a `table` parameter. Allowed values are positive integers starting from 1. Some values are already in use to refer to specific routing tables: see `/etc/iproute2/rt_tables`. (NetworkManager: as of v1.10.0)

- **mtu** (scalar) – since **0.101**

The MTU to be used for the route, in bytes. Must be a positive integer value.

- **congestion-window** (scalar) – since **0.102**

The congestion window to be used for the route, represented by number of segments. Must be a positive integer value.

- **advertised-receive-window** (scalar) – since **0.102**

The receive window to be advertised for the route, represented by number of segments. Must be a positive integer value.

- **routing-policy** (mapping)

The `routing-policy` block defines extra routing policy for a network, where traffic may be handled specially based on the source IP, firewall marking, etc.

For `from`, `to`, both IPv4 and IPv6 addresses are recognized, and must be in the form `addr/prefixlen` or `addr`.

- **from** (scalar)

Set a source IP address to match traffic for this policy rule.

- **to** (scalar)

Match on traffic going to the specified destination.

- **table** (scalar)

The table number to match for the route. In some scenarios, it may be useful to set routes in a separate routing table. It may also be used to refer to routes which also accept a `table` parameter. Allowed values are positive integers starting from 1. Some values are already in use to refer to specific routing tables: see `/etc/iproute2/rt_tables`.

- **priority** (scalar)

Specify a priority for the routing policy rule, to influence the order in which routing rules are processed. A higher number means lower priority: rules are processed in order by increasing priority number.

- **mark** (scalar)

Have this routing policy rule match on traffic that has been marked by the iptables firewall with this value. Allowed values are positive integers starting from 1.

- **type-of-service** (scalar)

Match this policy rule based on the type of service number applied to the traffic.

Authentication

Netplan supports advanced authentication settings for ethernet and wifi interfaces, as well as individual wifi networks, by means of the `auth` block.

- **auth** (mapping)

Specifies authentication settings for a device of type `ethernets:`, or an `access-points:` entry on a `wifis:` device.

The `auth` block supports the following properties:

- **key-management** (scalar)

The supported key management modes are `none` (no key management); `psk` (WPA with pre-shared key, common for home wifi); `eap` (WPA with EAP, common for enterprise wifi); and `802.1x` (used primarily for wired Ethernet connections).

- **password** (scalar)

The password string for EAP, or the pre-shared key for WPA-PSK.

The following properties can be used if `key-management` is `eap` or `802.1x`:
- **method** (scalar)

The EAP method to use. The supported EAP methods are `tls` (TLS), `peap` (Protected EAP), and `ttls` (Tunneled TLS).
- **identity** (scalar)

The identity to use for EAP.
- **anonymous-identity** (scalar)

The identity to pass over the unencrypted channel if the chosen EAP method supports passing a different tunnelled identity.
- **ca-certificate** (scalar)

Path to a file with one or more trusted certificate authority (CA) certificates.
- **client-certificate** (scalar)

Path to a file containing the certificate to be used by the client during authentication.
- **client-key** (scalar)

Path to a file containing the private key corresponding to `client-certificate`.
- **client-key-password** (scalar)

Password to use to decrypt the private key specified in `client-key` if it is encrypted.
- **phase2-auth** (scalar) – since **0.99**

Phase 2 authentication mechanism.

Properties for device type `ethernets`:

Ethernet device definitions, beyond common ones described above, also support some additional properties that can be used for SR-IOV devices.

- **link** (scalar) – since **0.99**

(SR-IOV devices only) The `link` property declares the device as a Virtual Function of the selected Physical Function device, as identified by the given netplan id.

Example:

```
ethernets:
  enp1: { ... }
  enp1s16f1:
    link: enp1
```

- **virtual-function-count** (scalar) – since **0.99**

(SR-IOV devices only) In certain special cases VFs might need to be configured outside of netplan. For such configurations `virtual-function-count` can be optionally used to set an explicit number of Virtual Functions for the given Physical Function. If unset, the default is to create only as many VFs as are defined in the netplan configuration. This should be used for special cases only.

Requires feature: `sriov`

- **embedded-switch-mode** (scalar) – since **0.104**

(SR-IOV devices only) Change the operational mode of the embedded switch of a supported SmartNIC PCI device (e.g. Mellanox ConnectX-5). Possible values are `switchdev` or `legacy`, if unspecified the vendor's default configuration is used.

Requires feature: eswitch-mode

- **delay-virtual-functions-rebind** (bool) – since **0.104**

(SR-IOV devices only) Delay rebinding of SR-IOV virtual functions to its driver after changing the `embedded-switch-mode` setting to a later stage. Can be enabled when bonding/VF LAG is in use. Defaults to `false`.

Requires feature: eswitch-mode

- **infiniband-mode** (scalar) – since **0.105**

(InfiniBand devices only) Change the operational mode of a IPoIB device. Possible values are `datagram` or `connected`. If unspecified the kernel's default configuration is used.

Requires feature: infiniband

Properties for device type modems:

GSM/CDMA modem configuration is only supported for the NetworkManager backend. `systemd-networkd` does not support modems.

Requires feature: modems

- **apn** (scalar) – since **0.99**

Set the carrier APN (Access Point Name). This can be omitted if `auto-config` is enabled.

- **auto-config** (bool) – since **0.99**

Specify whether to try and auto-configure the modem by doing a lookup of the carrier against the Mobile Broadband Provider database. This may not work for all carriers.

- **device-id** (scalar) – since **0.99**

Specify the device ID (as given by the WWAN management service) of the modem to match. This can be found using `mmcli`.

- **network-id** (scalar) – since **0.99**

Specify the Network ID (GSM LAI format). If this is specified, the device will not roam networks.

- **number** (scalar) – since **0.99**

The number to dial to establish the connection to the mobile broadband network. (Deprecated for GSM)

- **password** (scalar) – since **0.99**

Specify the password used to authenticate with the carrier network. This can be omitted if `auto-config` is enabled.

- **pin** (scalar) – since **0.99**

Specify the SIM PIN to allow it to operate if a PIN is set.

- **sim-id** (scalar) – since **0.99**

Specify the SIM unique identifier (as given by the WWAN management service) which this connection applies to. If given, the connection will apply to any device also allowed by `device-id` which contains a SIM card matching the given identifier.

- **sim-operator-id** (scalar) – since **0.99**

Specify the MCC/MNC string (such as “310260” or “21601”) which identifies the carrier that this connection should apply to. If given, the connection will apply to any device also allowed by `device-id` and `sim-id` which contains a SIM card provisioned by the given operator.

- **username** (scalar) – since **0.99**

Specify the username used to authenticate with the carrier network. This can be omitted if `auto-config` is enabled.

Properties for device type wifis:

Note that `systemd-networkd` does not natively support wifi, so you need `wpa_supplicant` installed if you let the `networkd` renderer handle wifi.

- **access-points** (mapping)

This provides pre-configured connections to NetworkManager. Note that users can of course select other access points/SSIDs. The keys of the mapping are the SSIDs, and the values are mappings with the following supported properties:

- **password** (scalar)

Enable WPA/WPA2 authentication and set the passphrase for it. If neither this nor an `auth` block are given, the network is assumed to be open. The setting

```
password: "S3kr1t"
```

is equivalent to

```
auth:
  key-management: psk
  password: "S3kr1t"
```

- **mode** (scalar)

Possible access point modes are `infrastructure` (the default), `ap` (create an access point to which other devices can connect), and `adhoc` (peer to peer networks without a central access point). `ap` is only supported with NetworkManager.

- **bssid** (scalar) – since **0.99**

If specified, directs the device to only associate with the given access point.

- **band** (scalar) – since **0.99**

Possible bands are `5GHz` (for 5GHz 802.11a) and `2.4GHz` (for 2.4GHz 802.11), do not restrict the 802.11 frequency band of the network if unset (the default).

- **channel** (scalar) – since **0.99**

Wireless channel to use for the Wi-Fi connection. Because channel numbers overlap between bands, this property takes effect only if the `band` property is also set.

- **hidden** (bool) – since **0.100**

Set to `true` to change the SSID scan technique for connecting to hidden WiFi networks. Note this may have slower performance compared to `false` (the default) when connecting to publicly broadcast SSIDs.

- **wakeonwlan** (sequence of scalars) – since **0.99**

This enables WakeOnWLAN on supported devices. Not all drivers support all options. May be any combination of `any`, `disconnect`, `magic_pkt`, `gtk_rekey_failure`, `eap_identity_req`, `four_way_handshake`, `rkill_release` or `tcp` (NetworkManager only). Or the exclusive `default` flag (the default).

- **regulatory-domain** (scalar) – since **0.105**

This can be used to define the radio’s regulatory domain, to make use of additional WiFi channels outside the “world domain”. Takes an ISO / IEC 3166 country code (like `GB`) or `00` to reset to the “world domain”. See [wireless-regdb](#) for available values.

Requires dependency: `iw`, if it is to be used outside the `networkd` (`wpa_supplicant`) backend.

Properties for device type bridges:

- **interfaces** (sequence of scalars)

All devices matching this ID list will be added to the bridge. This may be an empty list, in which case the bridge will be brought online with no member interfaces.

Example:

```
ethernets:
  switchports:
    match: {name: "enp2*"}
  [...]
bridges:
  br0:
    interfaces: [switchports]
```

- **parameters** (mapping)

Customization parameters for special bridging options. Time intervals may need to be expressed as a number of seconds or milliseconds: the default value type is specified below. If necessary, time intervals can be qualified using a time suffix (such as “s” for seconds, “ms” for milliseconds) to allow for more control over its behavior.

- **ageing-time, aging-time** (scalar)

Set the period of time to keep a MAC address in the forwarding database after a packet is received. This maps to the `AgeingTimeSec=` property when the `networkd` renderer is used. If no time suffix is specified, the value will be interpreted as seconds.

- **priority** (scalar)

Set the priority value for the bridge. This value should be a number between `0` and `65535`. Lower values mean higher priority. The bridge with the higher priority will be elected as the root bridge.

- **port-priority** (scalar)

Set the port priority to `.`. The priority value is a number between `0` and `63`. This metric is used in the designated port and root port selection algorithms.

- **forward-delay** (scalar)

Specify the period of time the bridge will remain in Listening and Learning states before getting to the Forwarding state. This field maps to the `ForwardDelaySec=` property for the networkd renderer. If no time suffix is specified, the value will be interpreted as seconds.

– **hello-time** (scalar)

Specify the interval between two hello packets being sent out from the root and designated bridges. Hello packets communicate information about the network topology. When the networkd renderer is used, this maps to the `HelloTimeSec=` property. If no time suffix is specified, the value will be interpreted as seconds.

– **max-age** (scalar)

Set the maximum age of a hello packet. If the last hello packet is older than that value, the bridge will attempt to become the root bridge. This maps to the `MaxAgeSec=` property when the networkd renderer is used. If no time suffix is specified, the value will be interpreted as seconds.

– **path-cost** (scalar)

Set the cost of a path on the bridge. Faster interfaces should have a lower cost. This allows a finer control on the network topology so that the fastest paths are available whenever possible.

– **stp** (bool)

Define whether the bridge should use Spanning Tree Protocol. The default value is “true”, which means that Spanning Tree should be used.

Properties for device type bonds:

- **interfaces** (sequence of scalars)

All devices matching this ID list will be added to the bond.

Example:

```
ethernets:
  switchports:
    match: {name: "enp2*"}
  [...]
bonds:
  bond0:
    interfaces: [switchports]
```

- **parameters** (mapping)

Customization parameters for special bonding options. Time intervals may need to be expressed as a number of seconds or milliseconds: the default value type is specified below. If necessary, time intervals can be qualified using a time suffix (such as “s” for seconds, “ms” for milliseconds) to allow for more control over its behavior.

– **mode** (scalar)

Set the bonding mode used for the interfaces. The default is `balance-rr` (round robin). Possible values are `balance-rr`, `active-backup`, `balance-xor`, `broadcast`, `802.3ad`, `balance-tlb`, and `balance-alb`. For Open vSwitch `active-backup` and the additional modes `balance-tcp` and `balance-slb` are supported.

– **lACP-rate** (scalar)

Set the rate at which LACPDU are transmitted. This is only useful in 802.3ad mode. Possible values are `slow` (30 seconds, default), and `fast` (every second).

- **mii-monitor-interval** (scalar)

Specifies the interval for MII monitoring (verifying if an interface of the bond has carrier). The default is `0`; which disables MII monitoring. This is equivalent to the `MIIMonitorSec=` field for the networkd backend. If no time suffix is specified, the value will be interpreted as milliseconds.
- **min-links** (scalar)

The minimum number of links up in a bond to consider the bond interface to be up.
- **transmit-hash-policy** (scalar)

Specifies the transmit hash policy for the selection of slaves. This is only useful in `balance-xor`, `802.3ad` and `balance-tlb` modes. Possible values are `layer2`, `layer3+4`, `layer2+3`, `encap2+3`, and `encap3+4`.
- **ad-select** (scalar)

Set the aggregation selection mode. Possible values are `stable`, `bandwidth`, and `count`. This option is only used in `802.3ad` mode.
- **all-slaves-active** (bool)

If the bond should drop duplicate frames received on inactive ports, set this option to `false`. If they should be delivered, set this option to `true`. The default value is `false`, and is the desirable behavior in most situations.
- **arp-interval** (scalar)

Set the interval value for how frequently ARP link monitoring should happen. The default value is `0`, which disables ARP monitoring. For the networkd backend, this maps to the `ARPIntervalSec=` property. If no time suffix is specified, the value will be interpreted as milliseconds.
- **arp-ip-targets** (sequence of scalars)

IPs of other hosts on the link which should be sent ARP requests in order to validate that a slave is up. This option is only used when `arp-interval` is set to a value other than `0`. At least one IP address must be given for ARP link monitoring to function. Only IPv4 addresses are supported. You can specify up to 16 IP addresses. The default value is an empty list.
- **arp-validate** (scalar)

Configure how ARP replies are to be validated when using ARP link monitoring. Possible values are `none`, `active`, `backup`, and `all`.
- **arp-all-targets** (scalar)

Specify whether to use any ARP IP target being up as sufficient for a slave to be considered up; or if all the targets must be up. This is only used for `active-backup` mode when `arp-validate` is enabled. Possible values are `any` and `all`.
- **up-delay** (scalar)

Specify the delay before enabling a link once the link is physically up. The default value is `0`. This maps to the `UpDelaySec=` property for the networkd renderer. This option is only valid for the `miimon` link monitor. If no time suffix is specified, the value will be interpreted as milliseconds.
- **down-delay** (scalar)

Specify the delay before disabling a link once the link has been lost. The default value is `0`. This maps to the `DownDelaySec=` property for the networkd renderer. This option is only valid for the `miimon` link monitor. If no time suffix is specified, the value will be interpreted as milliseconds.
- **fail-over-mac-policy** (scalar)

Set whether to set all slaves to the same MAC address when adding them to the bond, or how else the system should handle MAC addresses. The possible values are `none`, `active`, and `follow`.

– **gratuitous-arp** (scalar)

Specify how many ARP packets to send after failover. Once a link is up on a new slave, a notification is sent and possibly repeated if this value is set to a number greater than 1. The default value is 1 and valid values are between 1 and 255. This only affects `active-backup` mode.

For historical reasons, the misspelling `gratuitious-arp` is also accepted and has the same function.

– **packets-per-slave** (scalar)

In `balance-rr` mode, specifies the number of packets to transmit on a slave before switching to the next. When this value is set to 0, slaves are chosen at random. Allowable values are between 0 and 65535. The default value is 1. This setting is only used in `balance-rr` mode.

– **primary-reselect-policy** (scalar)

Set the reselection policy for the primary slave. On failure of the active slave, the system will use this policy to decide how the new active slave will be chosen and how recovery will be handled. The possible values are `always`, `better`, and `failure`.

– **resend-igmp** (scalar)

In modes `balance-rr`, `active-backup`, `balance-tlb` and `balance-alb`, a failover can switch IGMP traffic from one slave to another.

This parameter specifies how many IGMP membership reports are issued on a failover event. Values range from 0 to 255. 0 disables sending membership reports. Otherwise, the first membership report is sent on failover and subsequent reports are sent at 200ms intervals.

– **learn-packet-interval** (scalar)

Specify the interval between sending learning packets to each slave. The value range is between 1 and 0x7fffffff. The default value is 1. This option only affects `balance-tlb` and `balance-alb` modes. Using the `networkd` renderer, this field maps to the `LearnPacketIntervalSec=` property. If no time suffix is specified, the value will be interpreted as seconds.

– **primary** (scalar)

Specify a device to be used as a primary slave, or preferred device to use as a slave for the bond (i.e. the preferred device to send data through), whenever it is available. This only affects `active-backup`, `balance-alb`, and `balance-tlb` modes.

Properties for device type tunnels:

Tunnels allow traffic to pass as if it was between systems on the same local network, although systems may be far from each other but reachable via the Internet. They may be used to support IPv6 traffic on a network where the ISP does not provide the service, or to extend and “connect” separate local networks. Please see https://en.wikipedia.org/wiki/Tunneling_protocol for more general information about tunnels.

• **mode** (scalar)

Defines the tunnel mode. Valid options are `sit`, `gre`, `ip6gre`, `ipip`, `ipip6`, `ip6ip6`, `vti`, `vti6`, `wireguard` and `vxlan`. Additionally, the `networkd` backend also supports `gretap` and `ip6gretap` modes. In addition, the `NetworkManager` backend supports `isatap` tunnels.

• **local** (scalar)

Defines the address of the local endpoint of the tunnel. (For VXLAN) This should match one of the parent's IP addresses or make use of the networkd special values.

- **remote** (scalar)

Defines the address of the remote endpoint of the tunnel or multicast group IP address for VXLAN.

- **ttl** (scalar) – since **0.103**

Defines the Time To Live (TTL) of the tunnel. Takes a number in the range 1..255.

- **key** (scalar or mapping)

Define keys to use for the tunnel. The key can be a number or a dotted quad (an IPv4 address). For `wireguard` it can be a base64-encoded private key or (as of `networkd` v242+) an absolute path to a file, containing the private key (since 0.100). It is used for identification of IP transforms. This is only required for `vti` and `vti6` when using the `networkd` backend.

This field may be used as a scalar (meaning that a single key is specified and to be used for input, output and private key), or as a mapping, where you can further specify `input/output/private`.

- **input** (scalar)

The input key for the tunnel

- **output** (scalar)

The output key for the tunnel

- **private** (scalar) – since **0.100**

A base64-encoded private key required for WireGuard tunnels. When the `systemd-networkd` backend (v242+) is used, this can also be an absolute path to a file containing the private key.

- **keys** (scalar or mapping)

Alternate name for the key field. See above.

Examples:

```
tunnels:
  tun0:
    mode: gre
    local: ...
    remote: ...
    keys:
      input: 1234
      output: 5678
```

```
tunnels:
  tun0:
    mode: vti6
    local: ...
    remote: ...
    key: 59568549
```

```
tunnels:
  wg0:
    mode: wireguard
    addresses: [...]
    peers:
```

(continues on next page)

(continued from previous page)

```

- keys:
  public: rlbInAj0qV69CysWPQY7KEBnKxpYCpaWqOs/dLevdWc=
  shared: /path/to/shared.key
  ...
key: mNb70IIXTdGw4khM70FlzJ+UPs7lmcWHV7xjPgakMkQ=

```

```

tunnels:
  wg0:
    mode: wireguard
    addresses: [...]
    peers:
      - keys:
        public: rlbInAj0qV69CysWPQY7KEBnKxpYCpaWqOs/dLevdWc=
        ...
    keys:
      private: /path/to/priv.key

```

WireGuard specific keys:

- **mark** (scalar) – since **0.100**
Firewall mark for outgoing WireGuard packets from this interface, optional.
- **port** (scalar) – since **0.100**
UDP port to listen at or auto. Optional, defaults to auto.
- **peers** (sequence of mappings) – since **0.100**
A list of peers, each having keys documented below.

Example:

```

tunnels:
  wg0:
    mode: wireguard
    key: /path/to/private.key
    mark: 42
    port: 5182
    peers:
      - keys:
        public: rlbInAj0qV69CysWPQY7KEBnKxpYCpaWqOs/dLevdWc=
        allowed-ips: [0.0.0.0/0, "2001:fe:ad:de:ad:be:ef:1/24"]
        keepalive: 23
        endpoint: 1.2.3.4:5
      - keys:
        public: M9nt4YujIOmNrRmpIRTmYSfMdrpvE7u6WkG8FY8WjG4=
        shared: /some/shared.key
        allowed-ips: [10.10.10.20/24]
        keepalive: 22
        endpoint: 5.4.3.2:1

```

- **endpoint** (scalar) – since **0.100**
Remote endpoint IPv4/IPv6 address or a hostname, followed by a colon and a port number.
- **allowed-ips** (sequence of scalars) – since **0.100**

A list of IP (v4 or v6) addresses with CIDR masks from which this peer is allowed to send incoming traffic and to which outgoing traffic for this peer is directed. The catch-all 0.0.0.0/0 may be specified for matching all IPv4 addresses, and ::/0 may be specified for matching all IPv6 addresses.

– **keepalive** (scalar) – since **0.100**

An interval in seconds, between 1 and 65535 inclusive, of how often to send an authenticated empty packet to the peer for the purpose of keeping a stateful firewall or NAT mapping valid persistently. Optional.

– **keys** (mapping) – since **0.100**

Define keys to use for the WireGuard peers.

This field can be used as a mapping, where you can further specify the `public` and `shared` keys.

* **public** (scalar) – since **0.100**

A base64-encoded public key, required for WireGuard peers.

* **shared** (scalar) – since **0.100**

A base64-encoded preshared key. Optional for WireGuard peers. When the `systemd-networkd` backend (v242+) is used, this can also be an absolute path to a file containing the preshared key.

VXLAN specific keys:

• **id** (scalar) – since **0.105**

The VXLAN Network Identifier (VNI or VXLAN Segment ID). Takes a number in the range 1..16777215.

• **link** (scalar) – since **0.105**

netplan ID of the parent device definition to which this VXLAN gets connected.

• **type-of-service** (scalar) – since **0.105**

The Type Of Service byte value for a vxlan interface.

• **mac-learning** (scalar) – since **0.105**

Takes a boolean. When `true`, enables dynamic MAC learning to discover remote MAC addresses.

• **ageing, aging** (scalar) – since **0.105**

The lifetime of Forwarding Database entry learned by the kernel, in seconds.

• **limit** (scalar) – since **0.105**

Configures maximum number of FDB entries.

• **arp-proxy** (scalar) – since **0.105**

Takes a boolean. When `true`, bridge-connected VXLAN tunnel endpoint answers ARP requests from the local bridge on behalf of remote Distributed Overlay Virtual Ethernet (DOVE) clients. Defaults to `false`.

• **notifications** (sequence of scalars) – since **0.105**

Takes the flags `12-miss` and `13-miss` to enable netlink LLADDR and/or netlink IP address miss notifications.

• **short-circuit** (scalar) – since **0.105**

Takes a boolean. When `true`, route short circuiting is turned on.

- **checksums** (sequence of scalars) – since **0.105**

Takes the flags `udp`, `zero-udp6-tx`, `zero-udp6-rx`, `remote-tx` and `remote-rx` to enable transmitting UDP checksums in VXLAN/IPv4, send/receive zero checksums in VXLAN/IPv6 and enable sending/receiving checksum offloading in VXLAN.
- **extensions** (sequence of scalars) – since **0.105**

Takes the flags `group-policy` and `generic-protocol` to enable the “Group Policy” and/or “Generic Protocol” VXLAN extensions.
- **port** (scalar) – since **0.105**

Configures the default destination UDP port. If the destination port is not specified then Linux kernel default will be used. Set to 4789 to get the IANA assigned value.
- **port-range** (sequence of scalars) – since **0.105**

Configures the source port range for the VXLAN. The kernel assigns the source UDP port based on the flow to help the receiver to do load balancing. When this option is not set, the normal range of local UDP ports is used. Uses the form `[LOWER, UPPER]`.
- **flow-label** (scalar) – since **0.105**

Specifies the flow label to use in outgoing packets. The valid range is `0-1048575`.
- **do-not-fragment** (scalar) – since **0.105**

Allows setting the IPv4 Do not Fragment (DF) bit in outgoing packets. Takes a boolean value. When unset, the kernel’s default will be used.

Properties for device type vlans:

- **id** (scalar)

VLAN ID, a number between `0` and `4094`.
- **link** (scalar)

netplan ID of the underlying device definition on which this VLAN gets created.

Example:

```
ethernets:
  eno1: {...}
vlans:
  en-intra:
    id: 1
    link: eno1
    dhcp4: yes
  en-vpn:
    id: 2
    link: eno1
    addresses: [...]
```

Properties for device type `vrf`s:

- **table** (scalar) – since **0.105**
The numeric routing table identifier. This setting is compulsory.
- **interfaces** (sequence of scalars) – since **0.105**
All devices matching this ID list will be added to the vrf. This may be an empty list, in which case the vrf will be brought online with no member interfaces.
- **routes** (sequence of mappings) – since **0.105**
Configure static routing for the device; see the [Routing](#) section. The `table` value is implicitly set to the VRF's `table`.
- **routing-policy** (sequence of mappings) – since **0.105**
Configure policy routing for the device; see the [Routing](#) section. The `table` value is implicitly set to the VRF's `table`.

Example:

```

vrf:
  vrf20:
    table: 20
    interfaces: [ br0 ]
    routes:
      - to: default
        via: 10.10.10.3
    routing-policy:
      - from: 10.10.10.42
  [...]
bridges:
  br0:
    interfaces: []

```

Properties for device type `nm-devices`:

The `nm-devices` device type is for internal use only and should not be used in normal configuration files. It enables a fallback mode for unsupported settings, using the `passthrough` mapping.

Backend-specific configuration parameters

In addition to the other fields available to configure interfaces, some backends may require to record some of their own parameters in netplan, especially if the netplan definitions are generated automatically by the consumer of that backend. Currently, this is only used with `NetworkManager`.

- **networkmanager** (mapping) – since **0.99**
Keeps the `NetworkManager`-specific configuration parameters used by the daemon to recognize connections.
 - **name** (scalar) – since **0.99**
Set the display name for the connection.
 - **uuid** (scalar) – since **0.99**

Defines the UUID (unique identifier) for this connection, as generated by NetworkManager itself.

- **stable-id** (scalar) – since **0.99**

Defines the stable ID (a different form of a connection name) used by NetworkManager in case the name of the connection might otherwise change, such as when sharing connections between users.

- **device** (scalar) – since **0.99**

Defines the interface name for which this connection applies.

- **passthrough** (mapping) – since **0.102**

Can be used as a fallback mechanism to missing keyfile settings.

3.2 API specification

`libnetplan` is a component of the Netplan. project that contains the logic for data parsing, validation and generation. It is build as a dynamic `.so` library that can be used from different binaries (like Netplan's `generate`, `netplan-dbus`, the `netplan apply/try/get/set/...` CLI or via the corresponding Python bindings or external applications like the NetworkManager, using the Netplan backend).

3.3 Netplan CLI

Netplan's manpages describe the usage of the different command line interface tools available. Those are also installed on a system running Netplan and can be accessed, using the `man` utility.

3.3.1 Netplan CLI

`netplan-generate`

NAME

`netplan-generate` - generate backend configuration from netplan YAML files

SYNOPSIS

`netplan` [--debug] **generate** -h | --help

`netplan` [--debug] **generate** [--root-dir *ROOT_DIR*] [--mapping *MAPPING*]

DESCRIPTION

netplan generate converts netplan YAML into configuration files understood by the backends (**systemd-networkd(8)** or **NetworkManager(8)**). It *does not* apply the generated configuration.

You will not normally need to run this directly as it is run by **netplan apply**, **netplan try**, or at boot.

Only if executed during the systemd initializing phase (i.e. “Early bootup, before `basic.target` is reached”), will it attempt to start/apply the newly created service units. **Requires feature: generate-just-in-time**

For details of the configuration file format, see **netplan(5)**.

OPTIONS

-h, --help : Print basic help.

--debug : Print debugging output during the process.

--root-dir *ROOT_DIR* : Instead of looking in `{lib,etc,run}/netplan`, look in `/ROOT_DIR/{lib,etc,run}/netplan`

--mapping *MAPPING* : Instead of generating output files, parse the configuration files and print some internal information about the device specified in *MAPPING*.

HANDLING MULTIPLE FILES

There are 3 locations that netplan generate considers:

- `/lib/netplan/*.yaml`
- `/etc/netplan/*.yaml`
- `/run/netplan/*.yaml`

If there are multiple files with exactly the same name, then only one will be read. A file in `/run/netplan` will shadow - completely replace

- a file with the same name in `/etc/netplan`. A file in `/etc/netplan` will itself shadow a file in `/lib/netplan`.

Or in other words, `/run/netplan` is top priority, then `/etc/netplan`, with `/lib/netplan` having the lowest priority.

If there are files with different names, then they are considered in lexicographical order - regardless of the directory they are in. Later files add to or override earlier files. For example, `/run/netplan/10-foo.yaml` would be updated by `/lib/netplan/20-abc.yaml`.

If you have two files with the same key/setting, the following rules apply:

- If the values are YAML boolean or scalar values (numbers and strings) the old value is overwritten by the new value.
- If the values are sequences, the sequences are concatenated - the new values are appended to the old list.
- If the values are mappings, netplan will examine the elements of the mappings in turn using these rules.

SEE ALSO

netplan(5), **netplan-apply(8)**, **netplan-try(8)**, **systemd-networkd(8)**, **NetworkManager(8)**

netplan-apply

NAME

netplan-apply - apply configuration from netplan YAML files to a running system

SYNOPSIS

```
netplan [--debug] apply -h | --help
```

```
netplan [--debug] apply
```

DESCRIPTION

netplan apply applies the current netplan configuration to a running system.

The process works as follows:

1. The backend configuration is generated from netplan YAML files.
2. The appropriate backends (**systemd-networkd(8)** or **NetworkManager(8)**) are invoked to bring up configured interfaces.
3. **netplan apply** iterates through interfaces that are still down, unbinding them from their drivers, and rebinding them. This gives **udev(7)** renaming rules the opportunity to run.
4. If any devices have been rebound, the appropriate backends are re-invoked in case more matches can be done.

For information about the generation step, see **netplan-generate(8)**. For details of the configuration file format, see **netplan(5)**.

OPTIONS

-h, --help : Print basic help.

--debug : Print debugging output during the process.

KNOWN ISSUES

netplan apply will not remove virtual devices such as bridges and bonds that have been created, even if they are no longer described in the netplan configuration. That is due to the fact that netplan operates statelessly and is not aware of the previously defined virtual devices.

This can be resolved by manually removing the virtual device (for example `ip link delete dev bond0`) and then running **netplan apply**, by rebooting, or by creating a temporary backup of the YAML state in `/etc/netplan` before modifying the configuration and passing this state to netplan (e.g. `mkdir -p /tmp/netplan_state_backup/etc && cp -r /etc/netplan /tmp/netplan_state_backup/etc/` then running **netplan apply --state /tmp/netplan_state_backup**)

SEE ALSO

netplan(5), **netplan-generate(8)**, **netplan-try(8)**, **udev(7)**, **systemd-networkd.service(8)**, **NetworkManager(8)**

netplan-try

NAME

netplan-try - try a configuration, optionally rolling it back

SYNOPSIS

netplan [--debug] **try** -h | --help

netplan [--debug] **try** [--config-file *CONFIG_FILE*] [--timeout *TIMEOUT*]

DESCRIPTION

netplan try takes a **netplan(5)** configuration, applies it, and automatically rolls it back if the user does not confirm the configuration within a time limit.

A configuration can be confirmed or rejected interactively or by sending the SIGUSR1 or SIGINT signals.

This may be especially useful on remote systems, to prevent an administrator being permanently locked out of systems in the case of a network configuration error.

OPTIONS

-h, --help : Print basic help.

--debug : Print debugging output during the process.

--config-file *CONFIG_FILE* : In addition to the usual configuration, apply *CONFIG_FILE*. It must be a YAML file in the **netplan(5)** format.

--timeout *TIMEOUT* : Wait for *TIMEOUT* seconds before reverting. Defaults to 120 seconds. Note that some network configurations (such as STP) may take over a minute to settle.

KNOWN ISSUES

netplan try uses similar procedures to **netplan apply**, so some of the same caveats apply around virtual devices.

There are also some known bugs: if **netplan try** times out or is cancelled, make sure to verify if the network configuration has in fact been reverted.

As with **netplan apply**, a reboot should fix any issues. However, be sure to verify that the config on disk is in the state you expect before rebooting!

SEE ALSO

netplan(5), **netplan-generate(8)**, **netplan-apply(8)**

netplan-get

NAME

netplan-get - read merged netplan YAML configuration

SYNOPSIS

netplan [--debug] **get** -h | --help

netplan [--debug] **get** [--root-dir=ROOT_DIR] [key]

DESCRIPTION

netplan get [key] reads all YAML files from `/etc,lib,run/netplan/*.yaml` and returns a merged view of the current configuration

You can specify `all` as a key (the default) to get the full YAML tree or extract a subtree by specifying a nested key like: `[network.]ethernets.eth0`.

For details of the configuration file format, see **netplan(5)**.

OPTIONS

-h, --help : Print basic help.

--debug : Print debugging output during the process.

--root-dir : Read YAML files from this root instead of /

SEE ALSO

netplan(5), **netplan-set(8)**, **netplan-dbus(8)**

netplan-set

NAME

netplan-set - write netplan YAML configuration snippets to file

SYNOPSIS

```
netplan [--debug] set -h | --help
```

```
netplan [--debug] set [--root-dir=ROOT_DIR] [--origin-hint=ORIGIN_HINT] [key=value]
```

DESCRIPTION

netplan set [key=value] writes a given key/value pair or YAML subtree into a YAML file in `/etc/netplan/` and validates its format.

You can specify a single value as: "[network.]ethernets.eth0.dhcp4=[1.2.3.4/24, 5.6.7.8/24]" or a full subtree as: "[network.]ethernets.eth0={dhcp4: true, dhcp6: true}".

For details of the configuration file format, see **netplan(5)**.

OPTIONS

-h, --help : Print basic help.

--debug : Print debugging output during the process.

--root-dir : Write YAML files into this root instead of /

--origin-hint : Specify a name for the config file, e.g.: 70-netplan-set => /etc/netplan/70-netplan-set.yaml

SEE ALSO

netplan(5), **netplan-get(8)**, **netplan-dbus(8)**

Netplan provides a command line interface, called **netplan**, which a user can utilize to control certain aspects of the Netplan configuration.

Tool	Description
<code>help</code>	Show a generic help message
<code>info</code>	Show the available feature flags
<code>ip</code>	Query DHCP leases
<code>generate</code>	Generate backend specific configuration files from <code>/etc/netplan/*.yaml</code>
<code>apply</code>	Apply current netplan config to running system
<code>try</code>	Try to apply a new netplan config to running system, with automatic rollback
<code>get</code>	Get a setting by specifying a nested key like "ethernets.eth0.addresses", or "all"
<code>set</code>	Add new setting by specifying a dotted key=value pair like "ethernets.eth0.dhcp4=true"
<code>rebind</code>	Rebind SR-IOV virtual functions of given physical functions to their driver

3.4 Netplan D-Bus

Netplan provides a daemon that can be run to provide the `io.netplan.Netplan` D-Bus API, to control certain aspects of a system's Netplan configuration programmatically. See also: *DBus config API*.

3.4.1 netplan-dbus

NAME

`netplan-dbus` - daemon to access netplan's functionality via a DBus API

SYNOPSIS

`netplan-dbus`

DESCRIPTION

`netplan-dbus` is a DBus daemon, providing `io.netplan.Netplan` on the system bus. The `/io/netplan/Netplan` object provides an `io.netplan.Netplan` interface, offering the following methods:

- `Apply()` -> b: calls **netplan apply** and returns a success or failure status.
- `Generate()` -> b: calls **netplan generate** and returns a success or failure status.
- `Info()` -> a(sv): returns a dict "Features -> as", containing an array of all available feature flags.
- `Config()` -> o: prepares a new config object as `/io/netplan/Netplan/config/<ID>`, by copying the current state from `/etc,run,lib}/netplan/*.yaml`

The `/io/netplan/Netplan/config/<ID>` objects provide a `io.netplan.Netplan.Config` interface, offering the following methods:

- `Get()` -> s: calls **netplan get --root-dir=/run/netplan/config-ID all** and returns the merged YAML config of the the given config object's state
- `Set(s:CONFIG_DELTA, s:ORIGIN_HINT)` -> b: calls **netplan set --root-dir=/run/netplan/config-ID --origin-hint=ORIGIN_HINT CONFIG_DELTA**

`CONFIG_DELTA` can be something like: `network.ethernets.eth0.dhcp4=true` and `ORIGIN_HINT` can be something like: `70-snapd` (it will then write the config to `70-snapd.yaml`). Once `Set()` is called on a config object, all other current and future config objects are being invalidated and cannot `Set()` or `Try()/Apply()` anymore, due to this pending dirty state. After the dirty config object is rejected via `Cancel()`, the other config objects are valid again. If the dirty config object is accepted via `Apply()`, newly created config objects will be valid, while the older states will stay invalid.

- `Try(u:TIMEOUT_SEC)` -> b: replaces the main netplan configuration with this config object's state and calls **netplan try --timeout=TIMEOUT_SEC**
- `Cancel()` -> b: rejects a currently running `Try()` attempt on this config object and/or discards the config object
- `Apply()` -> b: replaces the main netplan configuration with this config object's state and calls **netplan apply**

For information about the `Apply()/Try()/Get()/Set()` functionality, see `netplan-apply(8)/netplan-try(8)/netplan-get(8)/netplan-set(8)` accordingly. For details of the configuration file format, see `netplan(5)`.

SEE ALSO

netplan(5), netplan-apply(8), netplan-try(8), netplan-get(8), netplan-set(8)

EXPLANATION

4.1 General structure & IDs

4.1.1 Introduction

Distribution installers, cloud instantiation, image builds for particular devices, or any other way to deploy an operating system put its desired network configuration into YAML configuration file(s). During early boot, the netplan “network renderer” runs which reads `{lib,etc,run}/netplan/*.yaml` and writes configuration to `/run` to hand off control of devices to the specified networking daemon.

- Configured devices get handled by `systemd-networkd` by default, unless explicitly marked as managed by a specific renderer (`NetworkManager`)
- Devices not covered by the network config do not get touched at all.
- Usable in `initramfs` (few dependencies and fast)
- No persistent generated config, only original YAML config
- Parser supports multiple config files to allow applications like `libvirt` or `lxd` to package up expected network config (`virbr0`, `lxdbr0`), or to change the global default policy to use `NetworkManager` for everything.
- Retains the flexibility to change backends/policy later or adjust to removing `NetworkManager`, as generated configuration is ephemeral.

4.1.2 General structure

netplan’s configuration files use the [YAML](#) format. All `{lib,etc,run}/netplan/*.yaml` are considered. Lexicographically later files (regardless of in which directory they are) amend (new mapping keys) or override (same mapping keys) previous ones. A file in `/run/netplan` completely shadows a file with same name in `/etc/netplan`, and a file in either of those directories shadows a file with the same name in `/lib/netplan`.

The top-level node in a netplan configuration file is a `network:` mapping that contains `version: 2` (the YAML currently being used by `curtin`, `MaaS`, etc. is version 1), and then device definitions grouped by their type, such as `ethernets:`, `modems:`, `wifis:`, or `bridges:`. These are the types that our renderer can understand and are supported by our backends.

Each type block contains device definitions as a map where the keys (called “configuration IDs”) are defined as below.

4.1.3 Device configuration IDs

The key names below the per-device-type definition maps (like `ethernets:`) are called “ID”s. They must be unique throughout the entire set of configuration files. Their primary purpose is to serve as anchor names for composite devices, for example to enumerate the members of a bridge that is currently being defined.

(Since 0.97) If an interface is defined with an ID in a configuration file; it will be brought up by the applicable renderer. To not have netplan touch an interface at all, it should be completely omitted from the netplan configuration files.

There are two physically/structurally different classes of device definitions, and the ID field has a different interpretation for each:

Physical devices

(Examples: ethernet, modem, wifi) These can dynamically come and go between reboots and even during runtime (hot plugging). In the generic case, they can be selected by `match:` rules on desired properties, such as name/name pattern, MAC address, driver, or device paths. In general these will match any number of devices (unless they refer to properties which are unique such as the full path or MAC address), so without further knowledge about the hardware these will always be considered as a group.

It is valid to specify no match rules at all, in which case the ID field is simply the interface name to be matched. This is mostly useful if you want to keep simple cases simple, and it’s how network device configuration has been done for a long time.

If there are `match:` rules, then the ID field is a purely opaque name which is only being used for references from definitions of compound devices in the config.

Virtual devices

(Examples: veth, bridge, bond, vrf) These are fully under the control of the config file(s) and the network stack. I. e. these devices are being created instead of matched. Thus `match:` and `set-name:` are not applicable for these, and the ID field is the name of the created virtual device.

4.2 Design

Network configuration abstraction via systemd-generator

4.3 FAQs

Find answers to common questions

Netplan is a network configuration abstraction renderer.

It is a **utility for network configuration** on a Linux system. You create a description of the required interfaces and define what each should do.

Netplan meets the need of **easy, descriptive network configuration** in YAML across a versatile set of server, desktop, cloud or IoT installations.

It is useful for **administrators of a Linux system** who want to use a common network configuration, controlling different backends like NetworkManager or systemd-networkd.

<i>Tutorials</i>	<i>How-to guides</i>
Get started - hands-on introduction to Netplan for new users	Step-by-step guides covering key operations and common tasks
<i>Explanation</i>	<i>Reference</i>
Discussion and clarification of key topics	Technical information - specifications, APIs, architecture

PROJECT AND COMMUNITY

Netplan is a member of the Ubuntu family. It's an open source project that warmly welcomes community contributions, suggestions, fixes and constructive feedback.

- **Read our code of conduct:** As a community we adhere to the Ubuntu code of conduct.
- **Get support:** Ask Ubuntu is a question and answer site for Ubuntu users and developers.
- **Join our online chat:** Meet us in #netplan on IRC Libera.Chat.
- **Report bugs:** We want to know about the problems so we can fix them.
- **Contribute code:** The code is open and we are open to accepting changes to it.

Thinking about using Netplan? [Get in touch!](#)