Netplan reference

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.

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:` , `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.

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.

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

Physical devices

(Examples: ethernet, wifi) These can dynamically come and go between reboots and even during runtime (hotplugging). 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) 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.

Common properties for physical device types

`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). Note that currently only networkd supports globbing, NetworkManager does not.

`macaddress` (scalar)

Device's MAC address in the form "XX:XX:XX:XX:XX:XX". Globs are not allowed.

`driver` (scalar)

Kernel driver name, corresponding to the `DRIVER` udev property. Globs are supported. 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
```

`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.

Common 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 `networks:`, for a device type (in e. g. `ethernets:`) or for a particular device definition. Default is `networkd`.

`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 autoconfiguration (SLAAC with RDNSS), this option can be set to cause the interface to be brought up. (Setting accept-ra alone is not sufficient.) Autoconfiguration will still honour 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-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.

Example to enable only IPv4 link-local: `link-local: [ ipv4 ]` Example to enable all link-local addresses: `link-local: [ ipv4, ipv6 ]` Example to disable all link-local addresses: `link-local: [ ]`

`critical` (bool)

(networkd backend only) Designate the connection as "critical to the system", meaning that special care will be taken by systemd-networkd to not release the IP from DHCP when the daemon is restarted.

`dhcp-identifier` (scalar)

When set to 'mac'; pass that setting over to systemd-networkd to use the device's MAC address as a unique identifier rather than a RFC4361-compliant Client ID. This has no effect when NetworkManager is used as a renderer.

`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)

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.

Example: `addresses: [192.168.14.2/24, "2001:1::1/64"]`

`gateway4`, `gateway6` (scalar)

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).

Example for IPv4: `gateway4: 172.16.0.1` Example for 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 ]
```

`routes` (mapping)

Configure static routing for the device; see the `Routing` section below.

`routing-policy` (mapping)

Configure policy routing for the device; see the `Routing` section below.

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.

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 higher metric on a preferred interface. Available for both the `networkd` and `NetworkManager` backends.

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.

`routes` (mapping)

The `routes` block defines standard static routes for an interface. At least `to` and `via` must be specified.

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.

`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.

`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), “unreachable”, “blackhole” or “prohibit”.

`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`.

`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.

Properties for device type `ethernets:`

Ethernet device definitions do not support any specific properties beyond the common ones described above.

Properties for device type `wifis:`

Note that `systemd-networkd` does not natively support wifi, so you need wpasupplicant 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 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.

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` (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`.

`lacp-rate` (scalar)

Set the rate at which LACPDUs 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. 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. 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 (ie. 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`, and `vti6`. 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.

`remote` (scalar)

Defines the address of the remote endpoint of the tunnel.

`key` (scalar or mapping)

Define keys to use for the tunnel. The key can be a number or a dotted quad (an IPv4 address). It is used for identification of IP transforms. This is only required for `vti` and `vti6` when using the networkd backend, and for `gre` or `ip6gre` tunnels when using the NetworkManager backend.

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

`input` (scalar)

The input key for the tunnel

`output` (scalar)

The output key for the tunnel

Examples:

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

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

`keys` (scalar or mapping)

Alternate name for the `key` field. See above.

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
    address: ...
```

Examples

Configure an ethernet device with networkd, identified by its name, and enable DHCP:

```
network:
  version: 2
  ethernets:
    eno1:
      dhcp4: true
```

This is an example of a static-configured interface with multiple IPv4 addresses and multiple gateways with networkd, with equal route metric levels, and static DNS nameservers (Google DNS for this example):

```
network:
  version: 2
  renderer: networkd
  ethernets:
    eno1:
      addresses:
      - 10.0.0.10/24
      - 11.0.0.11/24
      nameservers:
        addresses:
          - 8.8.8.8
          - 8.8.4.4
      routes:
      - to: 0.0.0.0/0
        via: 10.0.0.1
        metric: 100
      - to: 0.0.0.0/0
        via: 11.0.0.1
        metric: 100
```

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
  ethernets:
    # 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"
      gateway4: 192.168.14.1
      gateway6: 2001:1::2
      nameservers:
        search: [foo.local, bar.local]
        addresses: [8.8.8.8]
      routes:
        - 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
          table: 70
          priority: 50
      # only networkd can render on-link routes and routing policies
      renderer: networkd
    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
      # note: globbing is not supported by NetworkManager
      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
```

TABLE OF CONTENTS

Netplan

Home

Reference

Design

Examples

Need help?

Ask Ubuntu

Chat with Netplan on freenode

Report a bug

Contribute

GitHub

Legal info · Report a bug with this site · Report a bug with Netplan