# onway Mixer

# User Manual

2026-07-09

onway ag, Switzerland

# Contents

# 1 Introduction

onway Mixer is a configuration management application. It is designed to manage hundreds or thousands of device configurations efficiently and deploy these highly automated to the field. The change management focuses on reproducibility and keeping track of changes, even when working in distributed teams.

### Declarative Configuration

The configuration of all devices is maintained in a declarative form to completely represent the full device configuration state. Configuration declarations use two primary concepts: Device *lists* and device *templates*.

Device templates define a device configuration in a generic way, so a large set of devices implementing the same or similar use-cases can share the same device template. Templates can make use of other templates, so that the parts that multiple use-cases have in common can be shared in a single configuration.

To form a final device configuration, device lists are combined with templates. Each device defines a primary template to include, and a set of device specific variables. By substituting variables and including sub-templates, a final configuration is formed in a process called *rendering*.

### Git Versioning

Device configurations are all maintained in the *git* version control system. This allows a distributed team to make changes to templates and device lists. Also, any change applied to any device maintained in the system is versioned, allowing to completely track any changes or revert to previous configuration states. This makes configuration management completely transparent to any user participating.

A central git repository acts not only as a hub for exchanging changes between multiple users, but it also is the primary interface to interact with the Mixer system. The Mixer software acts as a git *update hook* to validate, render and deploy configuration changes. In fact, making changes to any device's configuration strictly requires to go through git revisions pushed to the Mixer system. This is fully by design, and ensures that any change is versioned, and the configuration state in the git repository always matches the configuration state deployed to systems in the field.

The Mixer workflow strictly aligns to the principles of git, and git concepts are used for developing and testing configurations: Making changes to device configurations usually is done in git branches; pushing non-master branches allows validating configurations, or even the deployment to a specific set of test devices to verify the configuration. Only if a developed configuration change is ready to get deployed to the production system, the change is merged to the master branch and deployed using a *git push*.

# 2 Architecture

The onway Mixer service usually runs on a dedicated (virtual) server, and provides an integral part of the larger onway product ecosystem. It serves as a central git repository, and users apply configuration changes via git commit pushes.

Users configuring Mobile Routers or VPN servers interface with the whole system through the git protocol by pushing commits, only. Device monitoring is out of scope in the context of the Mixer tooling.

### git Interaction

When receiving git pushes, the Mixer service validates the received changes. If validation fails, a git push is rejected, and the user has to correct its changes before being able to push them with success. Only valid configurations can be pushed to the Mixer service on any branch.

In addition to configuration validation, changes are also deployed to devices, depending on the branch name a user pushes, Usually, only pushes to the *master* branch are deployed, but this can be customized.

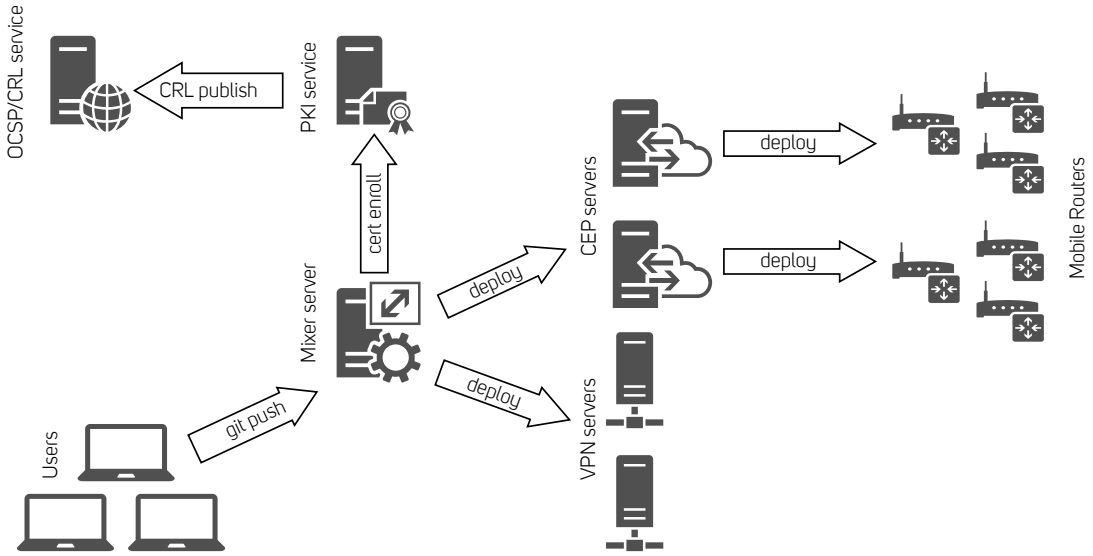
### User Tooling

Users interact with the Mixer configuration system through git, only. They may use any tool of choice to edit device lists and templates, or to commit changes to their local git repository clone. Basic git knowledge is required for users managing configurations, and only strict git commit management discipline by the users will reveal the full benefit of using the version control driven approach chosen by Mixer.

## Deployment Chain

For VPN server systems, the changes are deployed directly to the VPN host system. For Mobile Routers, deploying changes directly is infeasible, as changes may affect many routers, and not all of them may be reachable and online. Instead, Mobile Router configurations are deployed to the CEP server systems. All routers associated to a customer connect to this Customer Entry Point, and maintain a management connection. Configurations are synchronized to the Mobile Routers as soon as they become online.

Any number of VPN servers can be defined in a Mixer configuration. Multiple CEP servers are also supported, both to provide redundancy, but also to allow deploying CEP systems to different networks.




## Certificate Management

Issuing certificates is a complicated matter, and especially for large fleets requires appropriate tooling. The onway Mixer interfaces transparently with the onway PKI solution to issue certificates for Mobile Routers and VPN servers automatically. When decommissioning devices, the PKI also ensures proper revocation of certificates, so VPN servers can reject tunnel setup requests from routers, or vice-versa. CRL/OCSP serving is dedicated to a different server to allow different network policies for the PKI host with the sensitive Certificate Authority information and the publicly reachable CRL/OCSP service.

# 3 Setup

## 3.1 Installation

onway provides the Mixer software as a bundled Debian package for Ubuntu Server systems. The software runs under the unprivileged user `mixer`. The package creates the user and appropriate directories, including the shared git repository storing device configuration files. The software does not run as a service, but is invoked from a git hook. Hook setup for the repository is managed by the package. It is installed from a pre-configured onway package server using `apt install mixer`.

## 3.2 Configuration

The git repository and local configuration files are all stored under the `mixer` user home directory `/var/lib/mixer`. Users interacting with Mobile Router or VPN gateway configurations initially clone the git repository as the `mixer` Unix user using:

```
git clone mixer@<hostname>:/var/lib/mixer/mixer.git
```

### 3.2.1 User Management

To allow users to push commits to the Mixer repository, clients are authenticated using SSH keys. An initial administrator SSH public key must be installed manually to the `mixer` users `.ssh/authorized_keys` file. Further user and SSH key management may be handled through the Mixer repository itself, overwriting the `authorized_keys` file.

### 3.2.2 HTTP proxy configuration

Mixer fetches online resources, such as Mobile Router images and validation schemas, from `https://support.onway.ch`. By default it does this directly, but a HTTP proxy can be configured in the `proxy.yaml` configuration file. Currently, the file has a single option `server`, taking a `http://` URL for the HTTP proxy server to use. The URL may contain port and Basic Auth username and password information:

```
server: http://proxy.local:8888
```

### 3.2.3 Deployment host configuration

After creating Mobile Router and VPN gateway configurations, these are deployed to the target hosts using SSH. For Mobile Routers, the configurations are deployed to CEP servers for indirect configuration, whereas VPN gateway configurations are deployed to the VPN server hosts directly.

The SSH configuration parameters required to connect and deploy to these servers are currently maintained in a local configuration file on the Mixer host, outside of the git repository that holds the actual Router and VPN gateway configurations. The deployment host configuration is stored in `deployment.yaml`. The YAML file takes a list of CEPs and a list of VPN gateways to deploy to:

```
router:
  cep.xy.example.com:
    user: foo
    hostkey: ssh-rsa AAAAE2VjZHNhLXNoYT...
  cep.vw.example.com:
    user: bar
    port: 2222
    hostkey: ssh-rsa AAAAE2VjZHNhLXNoYT...
vpn:
  vpn01.xy.example.org:
    user: root
    hostkey: ssh-rsa AAAAE2VjZHNhLXNoYT...
```

The `router` section takes a set of CEP hostnames to deploy Mobile Router configurations to. Currently, all router configurations are deployed to all configured hosts under the `router` section. The optional `port` option may define a custom SSH port. The `user` option defines the SSH user to log in on the host. Authentication uses SSH private keys installed for the local `mixer` user; The server must be set up to allow connections using that key.

The `hostkey` option defines the SSH host key identifier of the SSH server to connect to, as stored in the `.ssh/known_hosts` file. It is required to establish trust in the server without interactively accepting the offered server host key.

Under the `vpn` section, a list of VPN gateway configurations take the SSH options to use for connecting to the VPN gateway host over SSH. The key defines the hostname of the server, and must match the VPN gateway host name as specified in the first column of the git-maintained VPN gateway CSV file.

### 3.2.4 Internal PKI

The devices managed in Mixer use certificates that are automatically issued by Mixer. One option to configure a Certificate Authority to issue such certificates is by configuring the issuing (Intermediate) CA certificate and the associated private key locally on the Mixer host.

Every subdirectory under the `mixer` home `cas` directory takes a CA configuration, where the subdirectory name defines the name of the CA, as used by device certificate configurations. The PEM encoded CA certificate file to issue certificates under must be stored in a file named `cert.pem`, whereas the unencrypted PEM encoded private key is stored in a file `key.pem`, both in the CA specific subdirectory:

```
cas/
  example-ca-1/
    cert.pem
    key.pem
  example-ca-2/
    cert.pem
    key.pem
```

### 3.2.5 External EST PKI

Instead of the Mixer internal CA, an external CA can be used to issue device certificates via the EST (Enrollment over Secure Transport) protocol. The configuration for EST CA services is stored in a file named `est-ca.yaml` in the `mixer` users home directory. The EST configuration has the following format:

```
example-ca-3:
  hostname: https://est.example.com:443
  username: example-user
  password: example-pw
  base-name: C=CH, O=Example Org
  trust-cert: |
    -----BEGIN CERTIFICATE-----
    ...
    -----END CERTIFICATE-----
```

The top level name defines the name of the CA, as referenced by device certificate configurations. It must not conflict with the CA name of any internal CA, as the external CA would prevail and override the internal one with the same name. `hostname` defines the EST server URL prefix to append the `.well-known` EST operation suffix to. `username` and `password` are used for HTTP Basic Authentication against the EST server. The `base-name` is used to construct the device Subject Distinguished Name in combination with the Common Name defined in the device certificate configuration. Under `trust-cert`, a root certificate in PEM format must be provided to establish trust in the EST server certificate.

# 4 File Formats

The full configuration state of all systems managed within Mixer is maintained in git using two different types of files in text format:

- *Device Lists* as CSV files
- *Templates* as YAML files

For each device defined in a device list, the primary template and potentially included sub-templates get rendered to the device configuration, usually with a set of per-device variables defined for that device in the CSV.

## 4.1 CSV Device Lists

Device lists must be stored in the git repository `csv` folder, and must have a `*.csv` file ending. All CSV files must use a Unix-style CSV dialect:

- Unix style newlines ( `\n` )
- Commas ( `,` ) as CSV delimiter
- A `"` quote char, using double-quoting to escape quote chars

CSV files may contain comments by placing a leading `#` character on a line. All CSV files must begin with a header line in the form `<kind>,Template[,<variable>[:ibl]]+`, where `<kind>` is one of:

- `router` for onway Mobile Router devices
- `vpn` for onway VPN gateways

The `Template` column defines the primary template file this device uses, potentially including other templates. The given per-device template takes a path to a YAML file relative to the `templates` directory in the git repository root.

### Device Variables

Zero or more `<variable>` columns define per-device variables to render into the template. Optionally, the variable name can be extended with a colon ( `:` ) to define one or more option characters for this variable. The following options are defined:

- `i` : The variable is rendered as integer value into the template
- `b` : The variable is rendered as boolean value into the template. Valid per-device values are `0` and `1` or `true` and `false` .
- `l` : The variable represents a list of values instead of just a value. The per-device CSV field separates list elements using the pipe character ( `|` ).

### CSV Examples

An example of a Mobile Router CSV device list is:

```
Router,Template,ip,ifnames:l,wifi-chan:i
NetModule/1800/00112B021E4A,mr.yaml,192.168.1.1,lan0|lan1,48
```

An example of a VPN Gateway CSV device list is:

```
VPN,Template,subnet,as-numbers:li,master:b
vpn.example.com,vpn.yaml,10.1.2.0/24,65002|65003|65004,true
```

## 4.2 YAML Template Files

All device configurations are rendered from templates; devices select their primary template in the CSV device list definition. All paths are relative to the `templates` directory under the git repository root and all templates must have a `*.yaml` file extension.

### Includes

Templates can include one more more additional templates using the `include` top-level YAML list defining the paths of template files to include. When *including* a template, the *included* template gets merged into the including template. All sections already defined in the including template get extended by the sections defined in the included template. Values of the including template have preference over values in included templates. YAML lists from an including template get extended with any items defined in the included template.

### Variables

In any YAML template string key or value, the full or a part of the string can be substituted by a variable. These variables can be defined per-device in the CSV or they can be defined in template files using the `vars` top-level YAML keyword. A `vars` section looks like this:

```yaml
vars:
  foo: bar
  net: 192.168.1.1/24
  pfx: 24
```

### Variable Substitution

A `<foo>` token in a YAML string value gets replaced by the value of the variable named `foo`. Additional text or multiple variables can be used in a YAML key or value, such as `foo-<bar>-<baz>`, rendering to `foo-1-test` if `bar` is `1` and `baz` is `test`.

In the device list CSV header column, the type of a variable can be specified with variable options. If a YAML value consists of a single variable only, without additional text (such as `<foo>`), the variable will be rendered not to a string, but to the same type as the variable. This is useful to produce non-string values from variables for configuration options that require specific value types. Note that this applies to YAML values, only, as YAML keys are always substituted as strings.

Variables can be defined multiple times, in the CSV and in templates. In such cases the definition from the CSV takes precedence, followed by the base template definition. A per-device or base template defined variable will be substituted in the include templates. On the

other hand a variable defined in an include template will not be substituted in the template which does the include.

### Functional Substitutions

In addition to `<foo>`-like variable substitution, a `<bar()>`-style syntax is used to call pre-defined helper functions to derive values for substitution. The called function may take arguments, and these arguments use YAML syntax like in an array. String arguments may use the variable substitution syntax to pass variables to functions. So the example

```
<bar("baz", 7, <foo>)>
```

calls the pre-defined helper function `bar()` with three arguments. The first argument is the string `baz`, the second the integer `7`, and the third the value of the variable `foo`, which can be a string or an integer. Template function arguments can be template variables or nested template functions, such as `<mul(<foo>, <add(3, <bar>)>)>`.

### Supported Template Functions

The following helper functions are defined:

| Function/Args | Description |
|---|---|
| `add(` | Calculates the sum of values `a` and `b`. |
| `a,` | First summand |
| `b)` | Second summand |
| Example: | `<add(3, 5)>` = 8 |
| `sub(` | Subtracts the value `b` from the value of `a`. |
| `a,` | Minuend |
| `b)` | Subtrahend |
| Example: | `<sub(5, 3)>` = 2 |
| `mul(` | Creates the product of value `a` and `b`. |
| `a,` | First factor |
| `b)` | Second factor |
| Example: | `<mul(2, 3)>` = 6 |
| `div(` | Divide the value of `a` by the value of `b`, rounding the value down to the next integer. |
| `a,` | Dividend |
| `b)` | Divisor |
| Example: | `<div(7, 3)>` = 2 |
| `mod(` | Calculates the remainder of the division of `a` by `b`. |
| `a,` | Dividend |
| `b)` | Modulus |
| Example: | `<mod(13, 3)>` = 1 |

| | |
|---|---|
| `netmap(` | Calculates the offset of the given `subnet` within the `src` block, and calculates the subnet at the same offset in the `dst` address block. Subnet allocation size in `src` and `dst` blocks may be different. Takes three subnets in CIDR format: |
| `subnet,` | A CIDR subnet instance to map from the `src` to the `dst` address block |
| `src,` | A start address of a address block to map from, and the size of each subnet allocated in this block, in CIDR notation |
| `dst)` | A start address of a destination block to map to, and the size of each subnet allocated in this block, in CIDR notation |
| Example: | `<netmap(10.1.2.0/24, 10.1.0.0/24, 172.16.0.0/16)>` The subnet size of `subnet` must always be the same as the subnet size in `src`, as `10.1.2.0/24` is an instance of a subnet in the block starting at `10.1.0.0` using `/24` subnet allocation. `10.1.2.0/24` is the subnet at offset `2` in the `src` block starting at `10.1.0.0`. In the `dst` block starting at `172.16.0.0`, the subnet at the same offset `2` with `/16`-sized subnets is `172.18.0.0/16`, which the example evaluates to. |

| | |
|---|---|
| `addrat(` | Gets an address `subnet` at the given `offset`, and returns that address. If the `mask` is given as `True`, the subnet size of the input subnet is included in the output address in CIDR notation. |
| `subnet,` | A subnet string in CIDR notation |
| `offset,` | An offset within the subnet to get the address for |
| `[mask])` | A boolean flag, to include subnet size in return value, defaults to `False` |
| Example: | `<addrat(192.168.1.0/24, 1, False)>` In the `subnet` `192.168.1.0/24`, the address at `offset` `1` is `192.168.1.1`. As the `mask` parameter is `False`, the subnet mask `/24` is omitted. The example evaluates to `192.168.1.1`. |

### Template Examples

Given a primary template

```
# templates/main.yaml
xyz:
  aint: 1
  astr: test
  alist: <baselist>
  be: <boo>
include:
- foo.yaml
- bar.yaml
vars:
  bla: blablabla
  boo: blabla
```

and the sub-templates

```
# templates/foo.yaml
xyz:
  aint: 2
  alist: [ 7, 8, <nine> ]
```

```
# templates/bar.yaml
xyz:
  aint: 2
  subobj:
    bla: <bla>
include:
- <inc>.yaml
```

```
# templates/baz.yaml
xyz:
  subobj:
    x: 2
  alist:
  - foo: <foo>
```

rendered with the CSV variables

```
Vpn,Template,inc,bla,foo,nine:i,baselist:li
vpn.example.com,main.yaml,baz,blu,bar,9,1|2|3
```

to the JSON representation:

```
"xyz": {
  "aint": 1,
  "astr": "test",
  "alist": [ 1, 2, 3, 7, 8, 9, {"foo": "bar"}],
  "be": "blabla",
  "subobj": { "bla": "blu", "x": 2 }
}
```

# 5 git Operations

The onway Mixer software is built around the git distributed version control system. Using git as a foundation allows distributed configuration change development and management. Branching and merging is a key concept of git, and the Mixer software encourages their use for effectively working with large configuration sets.

The *master* branch in a Mixer configuration repository always holds the configurations that are currently deployed to production systems, and pushing to the branch usually involves deployment. Making changes to the master branch must be done with caution, as it may affect critical devices. When working with templates, this may become difficult, as changing a single template may affect hundreds of devices, potentially with slightly different use-cases. Deploying such changes without prior testing may be infeasible, so the tooling must allow testing changes on some devices using a specific template, but not on others using the very same template.

## 5.1 Using Branches

By using branches, a user may create changes that are under development and/or testing, and are not yet ready to get pushed to the *master* branch. Pushing changes to the non-master branch does validate the configuration change, but does not deploy the change to devices affected by that change. Getting changes through Mixer validation is a first step in developing a change, and the user can easily do this by creating a custom branch and pushing it to the Mixer repository.

### Branch configurations

If explicitly configured to do so, changes caused by pushes to a non-master branch can be deployed to a limited set of (testing) devices. What devices are affected by pushes to which branches is controlled through the so-called *branch configuration*. This branch configuration is maintained in Mixer under the `meta` repository root directory in a file named `branches.yaml`.

The branch configuration file contains a section for branch names or branch name matching patterns. Under each branch, the `devices` list holds devices that are affected by a push to the specified branch. Depending on the branch configuration, the following actions are performed when pushing non-master branches:

| Has branch config | Is device listed | Do validation | Do deployment |
|---|---|---|---|
| No | Yes/No | Yes | No |
| Yes | No | No | No |
| Yes | Yes | Yes | Yes |

So if a pushed branch matches a branch configuration entry, a device configuration is validated and deployed only if it is listed under that entry.

The items given in the branch configuration are match patterns, they may match multiple branch names. *Shell Wildcard Patterns* are used to define the match, as specified by *fnmatch*[1]. Example:

```
'testing/*':
  devices:
  - vpn01.test.example.com
  - vpn02.test.example.com
development:
  devices:
  - NetModule/1800/00112B021E4A
```

The branch configuration consulted during a push is always the version from the branch that gets pushed. So specifying the branch configuration only in the specific branch is sufficient. However, the master branch also consults its branch configuration, and does not process devices that are listed in any section of any custom branch configuration. This implies that pushes the master branch may deploy devices that have a different configuration deployed by a non-master branch, unless the device is specifically assigned to a non-master branch in the branch configuration of the master branch.

**Note**

Listing devices in the branch configuration breaks one of the primary rules in Mixer, namely that the master branch always holds the configuration that is actually deployed to devices. This is intended and very useful for testing changes. However, this can also be confusing and a source of problems, so it is recommended to never list production systems in `meta/branches.yaml`.

For branch patterns in `branches.yaml`, the following additional options can be specified:

| Option | Type | Description | Default |
|---|---|---|---|
| `allow-force` | `boolean` | Allow pushing with `--force`, rewriting history | `true` |
| `allow-merges` | `boolean` | Allow pushing non-trivial merge commits | `true` |

If `allow-force` is set to `false`, pushing commits using `git push --force` will be rejected if it would rewrite the history of the branch. The `allow-merges` option controls whether non-trivial merge commits are allowed to be pushed to the branch. If set to `false`, only fast-forward merges are allowed, and pushing non-trivial merges will be rejected. It is usually recommended to set both options to `false` for the `master` branch, but they default to `true` for backwards compatibility.

### Managing Branches

Branches can be created with the normal git workflows. When pushing a new branch, Mixer will have to determine what devices are affected by a change. To do so, it will compare the new branch revision pushed against the current master branch. To get the expected results, branches shall always be based of the master branch.

Merging between branches has no special implications for Mixer: The commits that get pushed represent the state of the configuration. Deleting a branch does not trigger any actions in Mixer. Devices affected by changes deployed by the branch keep their state.

[1] `https://docs.python.org/3/library/fnmatch.html`

## 5.2 Using Tags

Tags can be set on revisions and pushed to the Mixer repository. Doing so validates the configuration associated to the tag, but does not trigger any deployment action.

## 5.3 Public Keys

To allow a user read/write access to the git repository a user needs to copy his public key into the `authorized_keys` file in the Mixer's `.ssh` directory. To simplify this task, a configuration file with all configured users is maintained in Mixer under the `meta` repository root directory in a file named `auth.yaml`. This implies that any user with access to the repository can set up access for other users or remove such permission, including those for the user itself. The user configuration file contains a section for `users`. Under this section all trusted users are listed with a user name and a `key`. The `key` contains the complete content of the public key file of a user. An example of such a configuration:

```
users:
  user:
    key: ssh-rsa AAAAB3NzaC1yc2EAAAA....
  anotheruser:
    key: ssh-rsa AAAAB3NzaC1yc2EAAAA....
```

**Note**

Manually added keys to the `authorized_keys` file will be deleted. Each time when a change on the `auth.yaml` occurs the `authorized_keys` file gets rewritten.

## 5.4 YAML linting

Mixer uses schemas to validate configurations and prevent a large class of configuration errors. In addition to schema validation, Mixer can perform a linting step on YAML files affected by a git push, and reject pushes violating a set of linter formatting rules.

Compared to schema validation, YAML linting requirements are specific to the YAML style used by a user group, and therefore is disabled by default. The linting configuration is maintained within the git repository root in the `.yamllint.yaml` configuration file. If the file exists on the pushed branch, YAML linting is performed. Maintaining the file in git allows users to use the same configuration for local linting using external tools or editors.

The linting rules in this configuration file can be freely defined according to the *yamllint*[2] documentation. Rules using the `warning` level produce a warning during git pushes, but only `error` level problems result in a rejected push operation.

[2] `https://yamllint.readthedocs.io/en/stable/configuration.html`

## 5.5 Push Options

To convey additional user input or settings to a Mixer run, a number of arguments are supported via git push options. One or multiple push options can be specified by calling `git push` with the `-o` flag and the corresponding option for every option to set.

There are two kinds of options, the *named argument*, which is of a key-value type, and the *flag*. With the named argument, a value is assigned to the option: `-o argument=value`. For the flag on the other hand, only the name of the option is specified to enable its effect: `-o flag`. If no value is assigned to a named argument, it will be treated like a flag option and be set to true. Mixer will reject syntactically incorrect push options, however unknown options with correct syntax will be silently ignored.

The currently supported push options are:

| Option | Type | Description |
|---|---|---|
| `loglevel` | Named | Set the verbosity level of the standard output logger. Levels: `debug`, `info`, `warning`, `error`, `critical` |
| `trace` | Flag | Enable exception tracing. If any exceptions occur, print the full Python traceback in addition to the simplified error trace printed by default. |
| `preview` | Flag/Named | Dump rendered device templates for inspection, but do not deploy anything. If given as a named argument with value `diff`, a unified diff is shown against the old template instead of the full new template. |
| `noreject` | Flag | Accept git push even if validation or deployment fails. |
| `reject` | Flag | Reject git push even if validation and deployment succeeds. |
| `carlos-schema` | Named | Use a custom schema version for Mobile Router configuration validation. |
| `vpnconf-schema` | Named | Use a custom schema version for VPN Gateway configuration validation. |

**Warning**

As the `noreject` option allows pushing commits for configurations that cannot be deployed, using the option may result in a mismatch between the configurations in the Mixer repository and the configuration actually used by devices. It should be used with care.

The schema version used for validating configurations is automatically selected from the software version a device is running. Overriding the schema version is mostly useful for developers when testing unreleased builds.

# 6 Client Tooling

Users are free to use the editor and tools of their preference for creating and editing YAML templates and CSV files. However, this chapter provides some recommendations that have been proven useful in configuration editing when using the freely available Visual Studio Code.

**Note**

The following tools are recommendations, but such third party software and their quality is not under control of onway. onway can not guarantee the correct behaviour of such software and can only provide limited support.

## 6.1 CSV Editing

Editing CSV files can be done using a full-blown spreadsheet application. CSV format and line endings must be considered when importing/exporting CSV files.

Alternatively, the Visual Studio Code *Rainbow CSV*[3] extension can be useful to edit CSV in text form. Alternatively, the *Edit csv*[4] extension supports editing CSV files in table form.

## 6.2 YAML Linting

As discussed in the *git Operations* section, YAML linting can be useful to enforce a common YAML style and prevent some common YAML formatting and structuring issues. The `.yamllint.yaml` configuration is maintained in the git repository, and therefore can be picked up by client side tools to provide immediate feedback about YAML linting issues to the user.

The recommended extension for Visual Studio Code is called *Linter*[5]. It additionally requires the installation of *yamllint*[6] to support linting YAML files.

[3] `https://marketplace.visualstudio.com/items?itemName=mechatroner.rainbow-csv`
[4] `https://marketplace.visualstudio.com/items?itemName=janisdd.vscode-edit-csv`
[5] `https://marketplace.visualstudio.com/items?itemName=fnando.linter`
[6] `https://yamllint.readthedocs.io/en/stable/quickstart.html#installing-yamllint`

## 6.3 YAML Schema Validation

In addition to linting, the Mixer enforces YAML schema validation when pushing commits. The same schema can be used in the client side editor to provide immediate feedback to the user about configuration validity and assist in editing configurations. Completion suggestions can be triggered in the default Visual Studio Code keyboard mapping using the `Ctrl` + `Space` key combination (`Cmd` + `i` on macOS).

The recommended extension for Visual Studio Code is called *YAML*[7]. To define the schema to use for template validation, the following Visual Studio Code configuration is required:

```
{
 "yaml.schemas": {
  "https://support.onway.ch/onway-router/schemas/mixer-ng/latest/mixer.yaml":
   "templates/*.yaml"
 }
}
```

It is recommended to maintain this configuration in git under `.vscode/settings.json`, so any user working on the repository can pick it up automatically.

**Note**

While software version specific schemas are provided and used in Mixer server side configuration validation, it is usually not practical to use version specific schemas on the client. Instead, the provided *latest* schema validates for the latest software version, and server side validation can catch any version specific issue during git pushes.

**Warning**

Configuration schemas define the syntax of *rendered* templates. When using variables or complex includes in templates, the un-rendered template may not properly validate against the schema. Client side validation can not currently handle this properly, nonetheless can the direct feedback be very useful to edit templates.

[7] `https://marketplace.visualstudio.com/items?itemName=redhat.vscode-yaml`

# 7 Common Configuration

## 7.1 Device Certificates

### Concept

The configuration format for a device defined through YAML templates depends on the type of the device: A Mobile Router device configuration uses different sections than the configuration for a VPN gateway. However, some configuration sections, such as the ones defined in this chapter, are generic and can be used for multiple device types.

To authenticate VPN tunnels or to establish WPA-Enterprise WiFi connections, a device may require X.509 certificates and associated private keys. These certificates can be issued automatically using a pre-installed CA during device deployment. The certificate enrolment will be performed automatically by the Mixer based on the respective user configuration. The desired configuration can be declared by the user in the device certificate section of the configuration. The Mixer will then automatically generate the device certificates and private keys based on the specified properties, sign it with the selected intermediate CA and deploy it.

As an alternative to internally generated certificates, external PKI servers are supported via the built-in EST (Enrollment over Secure Transport) interface. To enable this functionality, an EST configuration file needs to be set up by the mixer administrator. The configuration file can define multiple external PKI servers to connect to, and their names in the configuration can then be used as the CA name in device certificate configuration sections. The supported certificate options are the same as for the internal CA.

The device certificate configuration section uses a declarative and explicit approach. The certificate lifetime, defined through start of validity and expiration date, have to be defined by the user and will not be renewed automatically. If the optional starting date property is omitted, the certificate will be valid immediately after deployment. The expiration date property is not optional and always needs to be specified.

To renew a certificate, the validity starting date and the expiration date need to be adjusted in the configuration and pushed out. This allows the Mixer to generate the new, valid certificates and re-deploy them to the device.

> **Note**
>
> When issuing and deploying a certificate, the certificate chain from the EST CA is implicitly deployed, including any issuer certificate supplied via EST. However, changes to the certificate chain are not automatically rolled out. If any certificate in the EST CA is renewed or the chain is changed by other means, affected devices must renew their certificate to implicitly redeploy the changed certificate chain. Such a renewal can be triggered by changing the certificate expiration date.

### Certificate Options

The certificate section, named `certs`, is a list of certificate configuration entries. One such entry is composed of the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `common-name` | string | Certificate subject common name | Yes |

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `ca` | string | Intermediate CA to issue the certificate from | Yes |
| `not-before` | string/date | Certificate validity starting date | No |
| `not-after` | string/date | Certificate expiration date | Yes |
| `blocks` | list[string] | IPv4 or IPv6 CIDR subnets to include | No |
| `subject-altnames` | list[string] | Subject Alternative Names to use | No |

The given `common-name` is combined with all other relative Distinguished Names of the issuing CA to form the certificate subject. `not-before` / `not-after` dates optionally can include a time, and are in ISO-8601 format ( `YYYY-MM-DD [hh:mm:ss]` ) in UTC time. The `blocks` option lists subnets to include as RFC 3779 *IPAddrBlocks* that are assigned to the subject, allowing it to negotiate VPN tunnels bringing in those networks. The type of `subject-altnames` (mail address, DNS name, etc.) is determined automatically by the format the name is given in.

### Configuration Example

```
certs:
- common-name: some-device-name
  ca: some-CA-name
  not-before: 2022-06-08
  not-after: 2023-06-08 12:00:00
  blocks:
  - 1.2.3.0/24
  - fc00:1234::/64
  subject-altnames:
  - sub1.example.com
  - user@example.com
```

# 8 Mobile Router Specific Configuration

In addition to the `certs` section shared with other device types, a Mobile Router device takes additional configuration as specified in the following sections:

- `image` for software image selection
- `netconfig` for generic device and network configuration
- `telemetry` for telemetry data collection and reporting configuration
- `bundle` for filesystem bundle configuration

The `netconfig` and the `image` section are mandatory for a Mobile Router device configuration. The existence of these two sections is validated when the Mixer service receives git pushes. The `certs`, the `telemetry`, and the `bundle` sections are optional.

**Note**

While the telemetry section controls how telemetry is collected and reported, the Mixer does not consume or visualize collected telemetry data. If a setup is using *onway director*, it manages telemetry configuration implicitly. A potential telemetry configuration defined in Mixer is ignored.

Not all sections, options or values are supported by all versions of the Mobile Router image. If not otherwise marked, the configuration refers to image version `3.1.0`. Features requiring a newer image version use a version tag in the section, option or value description. Superscript is used to indicate version requirements, and $^{v3.2.0}$ indicates that at least image version `3.2.0` is required for using the feature. Mixer validation is version-aware, options not supported by the used version get rejected.

## 8.1 Software Image Version

A Mobile Router device configuration `image` tag defines the image release version a Mobile Router is intended to run. In a YAML template, `image` defines a string value of the image version, such as:

```
image: 3.1.0
```

The Mixer software automatically discovers the image file for the device target hardware using an image service provided by onway. The image file is downloaded and cached, and gets deployed automatically to the affected device. When changing the image version by pushing a git commit to a deploying branch, the image gets applied immediately to online routers. Once completed, the router immediately reboots into the new image, resulting in a short downtime.

**Note**

There are several safety measures in place to avoid breaking routers when deploying new image releases. Nonetheless, image updates, especially for major releases, shall be applied only after having been approved for a wider audience by onway and have been tested for the targeting use cases.

## 8.2 Networking Configuration

A Mobile Router's main configuration is provided in the `netconfig` section of its YAML template. It is used for network specific configuration, but includes some system related settings as well to define the behaviour of a router.

As with any other configuration done in Mixer, the `netconfig` is fully declarative. It consists of two subsections: `unconditional` for static configuration, and `conditional` for dynamic configuration fragments that are enabled only if certain system conditions are met.

Under both the `conditional` and `unconditional` sections, further subsections define various components of the router, the herein so-called *subsystems*. While each subsystem is configured on its own, only the combination of these subsystem sections form the larger use-case a Mobile Router is intended to implement.

For the `unconditional` section, *subsystem* configurations are directly nested, such as:

```
netconfig:
  unconditional:
    subsystem:
      # ...
    subsystem:
      # ...
```

The `conditional` section, however, activates a subsystem configuration only if a condition is fulfilled. `conditional` is a list, and each item uses an `if` - `then` - `else` construct for a condition:

```
netconfig:
  conditional:
  - if:
      condition:
        # ...
    then:
      subsystem:
        # ...
    else:
      subsystem:
        # ...
```

The `then` section defines subsystems to activate once the given condition is fulfilled, the `else` section defines a subsystem configuration to activate if the condition is not fulfilled. Both the `then` and `else` sections are optional.

Instead of a configuration fragment, the `then` and `else` sections can be defined as a list[v4.1.0], where each list item defines a nested condition, containing a `if` statement with optional `then` / `else` sections. Nested conditions in the `then` statement allows the *AND*-combination of conditions, while nesting in the `else` statement allows modelling *elseif*-functionality. Conditions can be nested to arbitrary levels.

```
  netconfig:
    conditional:
    - if:
        condition:
          # ...
      then:
      - if:
```

```
      condition:
        # ...
    then:
      subsystem:
        # ...
```

**Note**

The subsystem definition in conditionals does not have to stand on its own, but can extend or alter existing configurations from the `unconditional` section. The configuration part under a conditional is merged into the `unconditional` configuration tree; Values in the conditional have precedence over values from the unconditional configuration.

In the following two subsections, the various subsystems are introduced and defined, and so are the conditions to use for the `if` construct in conditionals.

## 8.3 Networking Configuration Subsystems

### 8.3.1 "addr" Subsystem

The `addr` subsystem installs and maintains IPv4 and IPv6 Layer 3 addresses with their associated network prefix on network interfaces. For each network interface, a list of strings takes addresses with their prefix in CIDR notation.

```
addr:
  lan0:
  - 192.168.0.1/24
  - fc00:1234::1/64
  wfi0:
  - 10.1.2.3/20
```

### 8.3.2 "api" Subsystem

The `api` subsystem configuration defines endpoints to provide the onway specific Mobile Router HTTP/REST API[8] under. The `api` section is a list of instances, each taking the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `port` | integer | TCP HTTP port to listen on | Yes |
| `ifname` | string | interface name to bind HTTP socket to | No |
| `handler` | object | Handlers to configure on this instance | Yes |

If `ifname` is omitted, the server listens on any interface in the default implicit VRF. The `handler` section takes additional subsections, each named after the API handler to allow, one of: `apc` v3.17.0, `cpu` v3.16.0, `gnss`, `imu`, `io` v4.5.0, `led`, `link` v3.16.0, `memory` v3.16.0, `modem`, `sms` v3.9.3, `storage` v3.16.0, `system`, `time` and `vpn` v3.9.0.

The following sections take additional handler specific options:

#### "gnss" handler

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `inject-wheel-speed` v4.2.0 | bool | Allow injecting external wheel speed | No |

If `inject-wheel-speed` is set to `True`, API clients may submit external wheel speed measurements to the GNSS receiver to improve dead reckoning performance. As this affects dead reckoning operation, the option must be explicitly enabled on the API instance providing access. It defaults to `False`.

[8] `https://support.onway.ch/onway-router/pubdoc/latest/router-api.html`

#### "io" handler

The `io` [v4.5.0] API handler exposes digital I/O pins. The pins to be exposed have to be provided as option. Each pin of the router can be exposed as read-only or read/write. Each pin is defined by an alias name which is used by the API user to address it. Pins defined as read/write have their default state set on startup.

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `get` | list[object] | List of pins to be exposed as read-only | No |
| `set` | list[object] | List of pins to be exposed as read/write | No |

For each aliased pin, following parameters are available:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `module` | string | Name of the module the pin is on | Yes |
| `pin` | string | Name of the pin of the module | Yes |

Furthermore the read/write pins (`set` list) have an additional mandatory parameter:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `default` | string | Default state of the pin, `low` or `high` | Yes |

Example:

```
api:
- port: 80
  handler:
    io:
      set:
        fan-error:
          module: g403-slot3
          pin: 24
          default: low
      get:
        vestibule:
          module: g403-slot3
          pin: 34
        bistro:
          module: g403-slot3
          pin: 42
    system: {}
- port: 8080
  ifname: vrf10
  handler:
    imu: {}
    gnss:
      inject-wheel-speed: True
```

### 8.3.3 "bridge" Subsystem

The `bridge` subsystem creates virtual network bridges in software, combining multiple Ethernet interfaces to a common Layer 2 domain to switch packets between them. Switching is automatically offloaded to hardware if supported for the bridged ports. Only Ethernet interfaces can be associated to a bridge, but this includes:

- Physical Ethernet interfaces
- The bridge interface of other `bridge` subsystem bridges
- WiFi interfaces from the `wifi` subsystem in AP or mesh (but not in client) mode
- VLAN interfaces from the `vlan` subsystem
- VXLAN interfaces from the `vxlan` subsystem
- VRF interconnects from the `vrf` subsystem
- (Un-)elevated interfaces from the `nac` subsystem
- Interfaces into containers from the `container` subsystem
- Interfaces into virtual machines from the `vm` subsystem

The `bridge` section takes subsections for a bridge interface to create and maintain. A slave interface associated to a bridge shall not be used by other subsystems; instead, the Layer 3 configuration is done on the bridge interface, and services also must be provided on the bridge interface. Each bridge interface section takes the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `slaves` | list[string] | Interfaces to associate to bridge | Yes |
| `isolate` | bool | Set to disable L2 forwarding between ports | No |

If the `isolate` flag is given and set to `True`, communication between hosts on different bridge ports is not permitted. Instead, clients may communicate with the Mobile Router on the Layer 3 addresses and the services provided on the bridge interface itself, only. To make interface names intuitive, bridge interfaces must start with `br`, followed by an arbitrary bridge number. Example:

```
bridge:
  br0:
    slaves:
    - lan0
    - wfi1
    - lan1.10
    isolate: True
```

### 8.3.4 "can" Subsystem

The `can` subsystem configures CAN interfaces for receiving CAN frames. The `can-data` telemetry source may then collect frames from such interfaces. It takes sections for multiple CAN interfaces to configure, each accepting the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `bitrate` | integer | Bitrate to operate the CAN interface with | Yes |
| `mode` | string | CAN socket mode, see table below | No |
| `options` | object | Additional options for the CAN socket | No |

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `forward` | list[string] | CAN frame forwarding destinations | No |

If the `mode` option is omitted, the specified CAN interface will only be configured with the given bitrate, brought up, and no frames will be received. This is useful for CAN interfaces that are not needed for telemetry but are used for other purposes, such as receiving CAN data in a container and/or a virtual machine.

| CAN mode | Description |
|---|---|
| `raw` | Received CAN frames are not interpreted. The CAN identifier and data fields are forwarded as-is. |
| `j1939` | Received CAN frames are interpreted as J1939 data. The PGN field and Broadcast Announce Messages are decoded and forwarded. The data is forwarded as-is. |
| `fms` v4.1.0 | Received CAN frames are interpreted as FMS data. A fine selection of PGNs are interpreted and made available for consumers and telemetry. |

The `options` v4.1.0 field is specific to the CAN interface mode. Following options are available:

**`fms` mode**

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `for-dead-reckoning` | bool | Use speed input for GNSS dead reckoning | No |

If the `for-dead-reckoning` option is set to `True`, the vehicle wheel based speed received over FMS is forwarded to the GNSS module for improved dead reckoning performance.

Regardless of the mode, all frames received on a CAN interface are forwarded unidirectionally to the interfaces listed in the `forward` v4.1.0 option. This is useful to forward frames to containers or virtual machines using virtual CAN interfaces. Example:

```
can:
  can0:
    mode: j1939
    bitrate: 250000
    forward:
    - vxcan0
    - vmcan1
```

### 8.3.5 "container" Subsystem

The `container` subsystem configures *Docker*-like containers to run with the Mobile Router embedded container runtime. It takes arbitrary named subsections with container names, each accepting the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `location` | string | Storage location identifier | Yes |

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `image` | string | Image name under storage location | Yes |
| `mem` | integer | memory limit for zram overlay, in bytes | Yes |
| `veth` | object | Virtual Ethernet links to connect into container | No |
| `vxcan` | object | VXCAN virtual CAN links to connect into container | No |
| `physif` | list[string] | Physical interfaces to use in container | No |
| `env` | list[string] | Environment variables to pass to container | No |
| `nodes` | list[string] | Device nodes the container has access to | No |
| `shares` | list[object] | Persistent shared folders | No |

The `location` option selects the storage location containing container images, which can be a filesystem bundle or removable storage. The `image` identifies an *Open Container Initiative* container image bundle. Such an image directory with the given name is located under the `container` root level folder on the storage device identified by `location`.

The `veth` option is a section with up to 16 key/value pairs, where each key defines an interface name on the Mobile Router network namespace, and the value an interface name in the container network namespace. The Mobile Router side interface must be named with `veth` and an arbitrary number, and can be configured using other subsystems. The container side interface configuration is up to the container implementation itself. The `physif` option takes a list of up to 16 interfaces on the Mobile Router, and moves them into the container; they disappear from the Mobile Router system while a container is running.

Similarly to the `veth` option, the `vxcan` [v4.1.0] option takes key/value pairs for VXCAN virtual CAN tunnel interfaces, where each key defines an interface name on the Mobile Router, and the value defines an CAN interface name in the container. The Mobile Router side interface must be named with `vxcan` and an arbitrary number. Referencing it in the `can` subsystem `forward` option allows the container to receive CAN frames on the container side CAN interface.

The `env` environment option takes a list of up to 64 additional environment variables to pass to the container executable and the hooks defined by the container image. Each entry is usually in the form `key=value`.

The `nodes` [v3.2.1] option takes a list of up to eight device nodes under `/dev` the container can request access to. The list of `shares` [v3.7.2] defines up to eight bind mounts to persistent storage to pass into the container, each taking the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `location` | string | Storage location identifier | Yes |
| `directory` | string | Name of shared directory in location | Yes |
| `mountpoint` | string | Container side mount point path | Yes |

On the specified storage `location`, a root folder named `shared` contains a `directory` to store shared files persistently under. The folder is created if it is missing. On the container side, that directory is accessible under the specified `mountpoint`, which must define an existing and absolute directory in the container root filesystem. Example:

```
container:
  mycontainer:
    location: BUNDLE:somestuff
    image: example
    mem: 268435456
    veth:
      veth0: eth0
    vxcan:
      vxcan0: can1
    physif:
    - can0
    env:
    - MYIP=192.168.1.10/24
    nodes:
    - ttyUSB0
    shares:
    - location: INTERNAL
      directory: myshare
      mountpoint: /mnt
```

### 8.3.6 "dhcp" Subsystem

The `dhcp` subsystem defines services using the DHCP protocol to acquire or provide dynamic IPv4 addressing information. Within the `dhcp` subsystem, the `client` section defines DHCP client configurations, the `server` section defines local DHCP server services, and the `relay` section defines DHCP relay agent configurations.

#### DHCP Client

The `client` section takes subsections for interfaces to configure via a DHCP client. For each interface, it takes the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `metric` | integer | Metric to install a default route with | No |
| `class-id` | string | DHCP Vendor-Class attribute to send | No |
| `user-class` | list[string] | DHCP User-Class attributes to send | No |
| `use-dns` | bool | Apply received DNS servers for interface | No |
| `hostname` | string | DHCP Host Name to send in requests | No |

If the `metric` option is given and the DHCP server provides a *Router* DHCP option, a default route is installed using the given route metric, as discussed in the `route` subsystem. Multiple DHCP clients must have distinct `metric` options. If the `use-dns` [v3.4.0] option is set to `True`, the DHCP client installs any received nameservers for use in DNS lookups over this interface, overriding the default nameservers. If the `hostname` [v3.19.0] is set, it is sent as DHCP Option 12 in requests. Otherwise the system hostname is sent in requests. Example:

```
dhcp:
  client:
    lan0:
      metric: 50
      class-id: foo
      use-dns: True
```

```
    user-class:
    - bar
    - baz
    hostname: my-host
```

#### DHCP Server

The `server` section takes subsections for interfaces to configure local DHCP server instances. For each interface, it takes the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `range` | list[string] | DHCP lease address range | Yes |
| `prefix` | integer | Network prefix to assign | No |
| `duration` | integer | DHCP lease duration, in seconds | No |
| `router` | string | Default gateway to assign | No |
| `nameserver` | list[string] | List of name server IP addresses | No |
| `ntpserver` | list[string] | List of NTP server IP addresses | No |
| `vendor` | object | Vendor specific options | No |
| `static` | list[object] | Static lease configuration | No |

The `range` option defines the range of IPv4 addresses to assign leases from. The list consists of two entries; the first defines the address range start, and the second the (inclusive) address range end. The `nameserver` option takes up to two nameservers to assign to DHCP clients. The `ntpserver` option[v3.18.2] takes up to four NTP servers to advertise to DHCP clients.

The `vendor` option is used to form a DHCP Option `125` with any sub-options to send in DHCP responses, see RFC 3925. In the configuration, it is an object whose keys specify an IANA allocated Private Enterprise Number (PEN), and the value is another object with its keys being vendor specific codes. Each value is an arbitrary string to encode under that PEN and code. If the value has a `0x` as prefix[v3.14.0], the value is encoded as binary, otherwise as string. The PEN and the code are keys and therefore must be strings, but must represent numbers in decimal or (with a `0x` prefix) in hexadecimal.

The `vendor` option can also be used to form a DHCP Vendor-Option `43` [v3.14.0] with any sub-options to send in DHCP responses, see RFC 2132. In the configuration, it is an object whose keys specify a DHCP Vendor-Class `60` string or a *fnmatch()*[9] pattern with `FNM_EXTMATCH` support. The value is another object with string keys being DHCP sub-options to encode in option `43`, given in decimal or in hexadecimal with a `0x` prefix. The sub-option value can be given as string or as binary with a `0x` prefix. When multiple Vendor Class Identifier patterns match, the sub-options for the longest matching pattern are selected.

If the `vendor` option is configured with a PEN, all DHCP responses include the DHCP Option `125`. If the `vendor` option is configured with a Vendor Class string or pattern of it, the DHCP Vendor-Option `43` options are only sent if the DHCP Vendor-Class `60` in the DHCP discover/request messages of the client matches the configured Vendor Class string.

The `static` [v3.11.0] option takes a list of static leases. A static lease consists of a `hostname` and the address `addr` to be assigned to a client including `hostname` as DHCP option 12 in DHCP

[9] `https://www.man7.org/linux/man-pages/man3/fnmatch.3.html`

requests. Instead of the `hostname` only, a combination of `hostname`, `circuit-id` [v3.17.0] and `remote-id` [v3.17.0] is allowed. At least one of the values must be used. The `circuit-id` and the `remote-id` options are matched against DHCP option 82 suboptions inserted by the DHCP relay agent. If the options are prefixed with a `0x`-prefix, the strings are interpreted in hexadecimal and the values are binary encoded. Static addresses may be within or outside of the specified `range`. Example:

```
dhcp:
  server:
    lan0:
      range:
      - 192.168.1.20
      - 192.168.1.99
      prefix: 24
      duration: 600
      router: 192.168.1.1
      nameserver:
      - 1.1.1.1
      - 8.8.8.8
      ntpserver:
      - 192.168.1.42
      - 192.168.1.88
      vendor:
        "39030":
          "0x1F": Foo
          "0xAB": Bar
        "42 Network Inc.":
          "1": 192.168.1.1,192.168.1.10
        "Cisco AP*":
          "0xF1": 0x0a0200050a030005 # two WLC IPv4 addresses in sub-option 0xF1
      static:
      - hostname: my-host
        addr: 192.168.1.21
      - hostname: my-other-host
        circuit-id: 0x1234
        remote-id: ABCDEEFG
        addr: 192.168.1.123
```

#### DHCP Relay

The `relay` section takes subsections for interfaces to configure DHCP relay helpers on, so that DHCP requests from clients on that interface get relayed to a dedicated DHCP server.

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `servers` | list[string] | List of IPv4 DHCP server addresses | No |
| `interfaces` | list[string] | List of interfaces to re-broadcast to | No |
| `circuit-id` | string | Circuit ID to insert | No |
| `remote-id` | string | Remote ID to insert | No |

The DHCP server to relay to can be specified using two methods: Using the `servers` option, a list of unicast DHCP server addresses can be specified, and any request message is forwarded to all of these addresses. Ordinary routing decisions define over which path the

request are sent, and sending them over VPN tunnels is supported. Alternatively (or additionally), the `interfaces` option takes a list of local interface names, where all requests from DHCP clients are re-broadcasted to. This allows forwarding of requests to another Layer 2 segment, even if the DHCP server on that segment is not directly known. The `circuit-id` v3.17.0 and the `remote-id` v3.17.0 can be used as part of DHCP option 82. A relay-agent can insert one or both of these values into the request to the DHCP server. If the value has a `0x`-prefix, the string is interpreted in hexadecimal and is converted to binary, otherwise the string is encoded as-is. Example:

```
dhcp:
  relay:
    lan0:
      servers:
      - 10.5.10.1
      - 10.5.20.2
    wfi0:
      interfaces:
      - lan2
      circuit-id: 0x1234
      remote-id: ABCDEEFG
```

**Note**

Crossing VRF domains for relay forwarding, in which the DHCP client is in a different VRF than the DHCP server, is not supported. Many DHCP clients use unicast addressing to renew existing leases, bypassing the relay used during the initial DHCP server discovery. Hence it is not helpful to separate the DHCP server into an isolated VRF domain, as a direct unicast communication between the client and the server must be possible for clients to renew their leases properly.

### 8.3.7 "dhcpv6" Subsystem

The `dhcpv6` subsystem defines the service using the DHCPv6 protocol to provide dynamic IPv6 addresses to clients. As DHCPv6 is not sufficient to fully configure clients, it is usually used in conjunction with the `radv` subsystem to assign routes via router advertisements. Within the `dhcpv6` subsystem, the `relay` section defines DHCPv6 relay agent configurations; DHCPv6 server or client mode is currently not supported.

#### DHCPv6 Relay

The `relay` section takes subsections for interfaces to configure DHCPv6 relay helpers on, so that DHCPv6 requests from clients on that interface get relayed to a dedicated DHCPv6 server.

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `servers` | list[string] | List of IPv6 DHCPv6 server addresses | Yes |

The DHCPv6 relay currently supports forwarding to DHCPv6 servers via unicast addressing, only. Forwarding requests via multicast is not supported. The same mechanisms for DHCPv6 relay unicast routing as for DHCPv4 apply. Example:

```
dhcpv6:
  relay:
    lan0:
      servers:
      - fc00:1234::10
```

### 8.3.8 "dns" Subsystem

The `dns` v3.9.0 subsystem defines services for the Domain Name System protocol.

#### DNS forwarder

The `dns` subsystem `forwarder` section takes configurations for DNS forwarding server instances. A DNS forwarder is a DNS server that is not a recursive resolver itself, but forwards DNS requests from clients to one or more recursive DNS servers. A forwarder instance is configured with a list of the following objects:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `ifname` | string | Interface to listen for DNS requests | No |
| `resolver` | list[object] | List of DNS resolvers to forward to | Yes |
| `static` | object | Static domain names to answer locally | No |

If an `ifname` is given, the DNS forwarder accepts client DNS requests only on this interface or in the specified VRF domain. If no `ifname` is given, DNS requests are accepted from any interface in the default VRF domain. Each `ifname` must be unique in the list of forwarders, and only one entry may exist omitting the `ifname` option.

Every `resolver` list item takes the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `server` | string | DNS resolver IPv4 or IPv6 address | Yes |
| `port` | integer | Custom DNS resolver UDP port | No |
| `source` | string | Source IPv4 or IPv6 address to send from | No |
| `ifname` | string | Network interface to forward DNS requests over | No |

The forwarder instance accepts DNS queries from clients on the listening interface, and forwards the DNS requests to one of the defined `resolver` servers. If an `ifname` is given, forwarded requests are sent from that interface, allowing to cross VRF domains when forwarding. The optional `source` v3.9.1 address can enforce the source address to send from in DNS requests towards the resolver. DNS responses received by the forwarder are relayed back to the client that originated the request, using the source address that the client sent the associated request to.

The `static` section takes objects with domain names to answer locally instead of forwarding the requests. Wildcards v3.11.0 for subdomains in the `static` entries are allowed with the syntax defined in RFC 4592. They are case insensitive. Every domain name object takes the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `A` | list[strings] | List of IPv4 addresses for A queries | No |
| `AAAA` | list[strings] | List of IPv6 addresses for AAAA queries | No |

If a client request is for a defined static domain name, but no resource record is defined for the requested type, an empty response is sent to the client. Example:

```
dns:
  forwarder:
  - ifname: lan0
    resolver:
    - server: 8.8.8.8
      ifname: lan1
    - server: 9.9.9.9
      port: 9953
    static:
      test.example.com:
        A: [ 10.0.0.1, 10.0.0.2 ]
      "*.foo.example.com":
        A: [ 10.0.1.2 ]
```

#### Local static DNS records

The `dns` subsystem `static` [v3.14.0] section configures static DNS records to serve for DNS requests from local services on the the Mobile Router itself. It does not affect DNS requests from clients on the network.

The `static` section takes objects with domain names to answer locally instead of forwarding the requests. Wildcards for subdomains are allowed with the syntax defined in RFC 4592. They are case insensitive. Every domain name object takes the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `A` | list[strings] | List of IPv4 addresses for A queries | No |
| `AAAA` | list[strings] | List of IPv6 addresses for AAAA queries | No |

If a local DNS request is for a defined static domain name, but no resource record is defined for the requested type, an empty response is sent to the client. Example:

```
dns:
  static:
    test.example.com:
      A: [ 10.0.0.1, 10.0.0.2 ]
    "*.foo.example.com":
      A: [ 10.0.1.2 ]
```

### 8.3.9 "`firewall`" Subsystem

The `firewall` subsystem configuration defines custom firewall rules for filtering and Network Address Translation (NAT) to install on the router, for both IPv4 and IPv6 address families. It has nested subsections for different purposes, namely:

| Rule Type | Description |
|---|---|
| `forward` | filtering traffic that is forwarded (routed on Layer 3) on the local system |
| `input` | filtering traffic that is to receive by the local system itself |
| `snat` | translating source addresses (Source NAT) |
| `dnat` | translating destination addresses (Destination NAT) |
| `masq` | translating source addresses of outbound traffic to a local address dynamically (Masquerading) |
| `netmap` | translating addresses using a 1:1 network mapping |
| `policy` | Set global firewall policies |

#### Forwarding Rules

The `forward` rule set defines rules to allow traffic forwarding for, that is, enable routing of IP traffic. If no rule is specified matching the traffic to route, no forwarding is done, and the traffic is dropped instead. However, there are a few exceptions where forwarding is enabled by default:

- ICMP (for IPv4) and ICMPv6 traffic is always forwarded within the same VRF domain. ICMP is considered a requirement to properly operate and diagnose a network.
- Traffic subject to IPsec VPN tunnels or coming from such a tunnel, as configured in the `vpn` subsystem, is implicitly allowed. Tunnelling usually involves forwarding, and (negotiated) tunnel selectors are more comprehensive in defining traffic to allow over a tunnel.
- Traffic allowed in one direction is allowed in the reverse direction for the same connection (stateful firewall).

For any other traffic that shall be routed, explicit routes are required. The `forward` section is a list of rules, each taking the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `iif` | string | Input interface to match | No |
| `oif` | string | Output interface to match | No |
| `src` | string | CIDR subnet source address to match | No |
| `dst` | string | CIDR subnet destination address to match | No |
| `udp` / `tcp` | object | Layer 4 protocol match | No |
| `icmp` / `icmp6` | object | ICMP protocol match | No |

A `udp` or `tcp` option is a section, and takes the following suboptions to further define the Layer 4 protocol match:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `src` | integer | Source port to match | No |
| `src-range` | list[integer] | Range of source ports to match | No |
| `dst` | integer | Destination port to match | No |
| `dst-range` | list[integer] | Range of destination ports to match | No |

To match specific ports, the `src` and `dst` options can be specified to match a single port. Alternatively[v3.17.0], `src-range` and `dst-range` can be used, each taking a list with exactly two port numbers, specifying the start and end of an inclusive port range to match.

For `icmp` and `icmp6` protocol matches, the following suboptions can be used to limit the match to ICMP and ICMPv6 protocol fields:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `type` | integer | ICMP message type to match | No |
| `code` | integer | ICMP message code to match | No |

Example:

```
firewall:
  forward:
  - iif: lan0
    oif: lan1
    src: 10.0.1.0/24
    dst: 10.0.2.1/32
    tcp:
      src-range:
      - 1024
      - 65535
      dst: 80
  - iif: ivr0
    icmp:
      type: 8
```

#### Input Rules

The `input` rule set defines rules to allow traffic targeting the Mobile Router itself. It is a list of rules, each taking the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `iif` | string | Input interface to match | No |
| `udp` / `tcp` | object | Layer 4 protocol match | No |

Under the Layer 4 protocol match, the `dst` option takes an integer port number to open for incoming traffic.

**Note**

Specifying explicit `input` rules is usually not required for services defined in any of the other subsystems; each service sets up firewall rules implicitly to allow traffic from clients using that service. An exception is SSH access to a Mobile Router, which is refused from local interfaces by default, but can be enabled using appropriate firewall rules.

#### Source NAT Rules

The `snat` rule set defines rules for Source based Network Address Translation. SNAT rules do the translation only; when applying SNAT to forwarded traffic, a forward rule may be required, matching to traffic in a view before the translation. SNAT rules currently work for the IPv4 address family, only. `snat` is a list of rules, each taking the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `oif` | string | Output interface to match | No |
| `src` | string | CIDR subnet source address to match | No |
| `dst` | string | CIDR subnet destination address to match | No |
| `udp` / `tcp` | object | Layer 4 protocol information | No |
| `to` | string | CIDR subnet to map source to | Yes |

With the exception of `to`, all options define matches to what traffic to apply the SNAT rule to. The `to` option defines a subnet to select source addresses from to use as translated address. That subnet can be of different size than `src`, and can even be a single address. The `udp` and `tcp` sections take these options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `src` | integer | Source port to match | No |
| `src-range` | list[integer] | Range of source ports to match | No |
| `dst` | integer | Destination port to match | No |
| `dst-range` | list[integer] | Range of destination ports to match | No |
| `to` | integer | Port to translate source port to | No |
| `to-range` | list[integer] | Range of ports to translate source port to | No |

To match specific ports, the `src` and `dst` options can be specified to match a single port. Alternatively[v3.17.0], `src-range` and `dst-range` can be used, each taking a list with exactly two port numbers, specifying the start and end of an inclusive port range to match.

If a `to` option is defined in the Layer 4 protocol definition, the rule translates to an explicit source port in addition to the Layer 3 source address translation. Alternatively, a `to-range`[v3.17.0] takes a list of exactly two port numbers, specifying the start and end of an inclusive port range to randomly translate source ports to.

> **Note**
>
> SNAT is performed in `POSTROUTING` as a very last step before a packet leaves the system. When performing SNAT on traffic that crosses multiple VRF domains, it is recommended to perform SNAT in the last VRF domain using an `oif` match on the interface the packet leaves the system.

Example:

```
firewall:
  snat:
  - oif: lan0
    src: 10.1.1.0/24
    tcp:
      src-range:
```

```
    - 1
    - 1024
    dst: 80
    to: 8080
  to: 192.168.1.1/32
```

#### Destination NAT Rules

The `dnat` rule set defines rules for Destination based Network Address Translation. DNAT rules do the translation only; when applying DNAT to forwarded traffic, a forward rule may be required, matching to traffic in a view after the translation. DNAT rules currently work for the IPv4 address family, only. `dnat` is a list of rules, each taking the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `iif` | string | Input interface to match | No |
| `src` | string | CIDR subnet source address to match | No |
| `dst` | string | CIDR subnet destination address to match | No |
| `udp` / `tcp` | object | Layer 4 protocol information | No |
| `to` | string | CIDR subnet to map destination to | Yes |

With the exception of `to`, all options define matches to what traffic to apply the DNAT rule to. The `to` option defines a subnet to select destination addresses from to use as translated address. That subnet can be of different size than `dst`, and can even be a single address. The `udp` and `tcp` sections take these options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `src` | integer | Source port to match | No |
| `src-range` | list[integer] | Range of source ports to match | No |
| `dst` | integer | Destination port to match | No |
| `dst-range` | list[integer] | Range of destination ports to match | No |
| `to` | integer | Port to translate destination port to | No |
| `to-range` | list[integer] | Range of ports to translate destination port to | No |

To match specific ports, the `src` and `dst` options can be specified to match a single port. Alternatively[v3.17.0], `src-range` and `dst-range` can be used, each taking a list with exactly two port numbers, specifying the start and end of an inclusive port range to match.

If a `to` option is defined in the Layer 4 protocol definition, the rule translates to an explicit destination port in addition to the Layer 3 destination address translation. Alternatively, a `to-range`[v3.17.0] takes a list of exactly two port numbers, specifying the start and end of an inclusive port range to translate source ports to. If a `dst-range` and a `to-range` define a range with the same number of ports, the ports are translated using a one-to-one offset mapping, so that the first port in the `dst-range` is translated to the first port in the `to-range`, the second port in the `dst-range` to the second port in the `to-range`, and so on. If the ranges have different sizes, the translation is done randomly.

**Note**

DNAT is performed in `PREROUTING` as a very first step before a packet enters the system. When performing DNAT on traffic that crosses multiple VRF domains, it is recommended to perform DNAT in the first VRF domain using an `iif` match on the interface the packet enters the system.

Example:

```
firewall:
  dnat:
  - iif: br0
    dst: 10.1.0.0/16
    to: 192.168.1.0/24
    udp:
      dst-range:
      - 1600
      - 1699
      to-range:
      - 21600
      - 21699
```

#### Masquerading Rules

The `masq` rule set defines rules for dynamic source NAT, where the source address is translated to the local address of an interface where traffic leaves the system. This is less flexible than plain `snat`, but can work with dynamic addresses on that interface, such as those acquired over DHCP. Masquerading is currently applied to IPv4 traffic, only. The `masq` option is a list of rules, each taking the following option:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `oif` | string | Output interface to masquerade on | Yes |

Example:

```
firewall:
  masq:
  - oif: mbm0
  - oif: mbm1
```

#### Netmap Rules

The `netmap` rule set defines rules for bi-directional Network Address Translation using a 1:1 mapping with full subnets. Each rule defines symmetrical mapping in both directions, doing Source NAT in one direction and Destination NAT in the other, so traffic can be initiated from either end. It currently works for IPv4 addresses only, and each rule in the `netmap` list takes the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `from` | string | CIDR subnet of address to map from | Yes |
| `to` | string | CIDR subnet of address to map to | Yes |
| `match` | string | CIDR subnet of address to match against | No |

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `ifname` | string | Interface name to match | No |

The `from` option specifies a subnet of source addresses of the packets that get source-NATed. The `to` option specifies a CIDR subnet to map that source address when doing source NAT, and must have the same subnet size as `from`. The `match` option defines the destination address to match against, and the `ifname` option the output interface, both in the source-NAT direction. In the reverse destination-NAT direction, the `to` option defines the packets to de-NAT into the network of the `from` option. The `match` option defines the source address to match, and the `ifname` option the input interface, both in the destination-NAT direction. Matching the input interface in source-NAT direction is not supported. Example:

```
firewall:
  netmap:
  - from: 10.0.1.0/24
    to: 192.168.1.0/24
    match: 10.7.0.0/16
    ifname: lan0
```

#### Firewall Policies

The `policy` v3.18.0 rule allows to change default system policies.

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `icmp-forward` | string | ICMP forward option to apply | No |

The `icmp-forward` option allows to change the ICMP forwarding policy. The `drop` policy drops any ICMP echo request in the FORWARDING chain but allows all other ICMP messages. The `allow` policy allows all ICMP messages, both within VRFs and when crossing VRFs over `ivr` devices. If the option `icmp-forward` is omitted, the default policy applies. Any ICMP messages within the same VRF are allowed and it drops unrelated ICMP traffic when crossing VRFs through `ivr` devices.

```
firewall:
  policy:
    icmp-forward: allow
```

### 8.3.10 "gnss" Subsystem

The `gnss` subsystem specifies GNSS related options.

#### Proxy section

The `proxy` v3.9.0 section allows to configure NMEA TCP proxies and NMEA UDP stream services.

The `tcp` subsection configures NMEA TCP proxies to allow networked clients to receive raw NMEA data for positioning. It takes a list of instances to create using these options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `port` | integer | TCP port to listen for client connects | Yes |
| `ifname` | string | Interface to bind TCP listen socket to | No |

If the `ifname` option is omitted, the socket listens on any interface in the implicit default VRF domain.

The `udp` subsection configures NMEA UDP stream services to send positioning streams to configured clients. It takes a list of instances to create using these options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `dst` | string | Destination IP address to send stream to | Yes |
| `port` | integer | UDP destination port to send stream to | Yes |
| `ifname` | string | Interface to send UDP stream over | No |

If the `ifname` option is omitted, the stream is routed within the implicit default VRF.

If a device has multiple GNSS receivers, the data of the best receiver is passed to clients. Example:

```
gnss:
  proxy:
    tcp:
    - port: 10110
      ifname: vrf10
    udp:
    - dst: 10.2.4.3
      port: 10110
      ifname: vrf11
```

### 8.3.11 "`http`" Subsystem

The `http` subsystem configures onway Media Server[10] instances, a HTTP web server serving static or template-rendered content to clients. It takes a list of instances, each with the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `port` | integer | TCP port to listen for client connects | Yes |
| `ifname` | string | Interface to bind TCP listen socket to | No |
| `location` | string | Storage location identifier | Yes |
| `type` | string | Storage backend, `removable` or `bundle` | No[v3.8.0] |

The `location` option selects the storage location containing files to serve, which can be a filesystem bundle or removable storage. The `type` defines the type of storage used, which is `removable` for removable media such as USB sticks, or `bundle` for filesystem bundles. If the `ifname` option is omitted, the web server instance listens on any interface in the implicit default VRF. Example:

[10] `https://support.onway.ch/onway-router/pubdoc/latest/media-server.html`

```
http:
- port: 80
  ifname: lan0
  location: USB:1
  type: removable
```

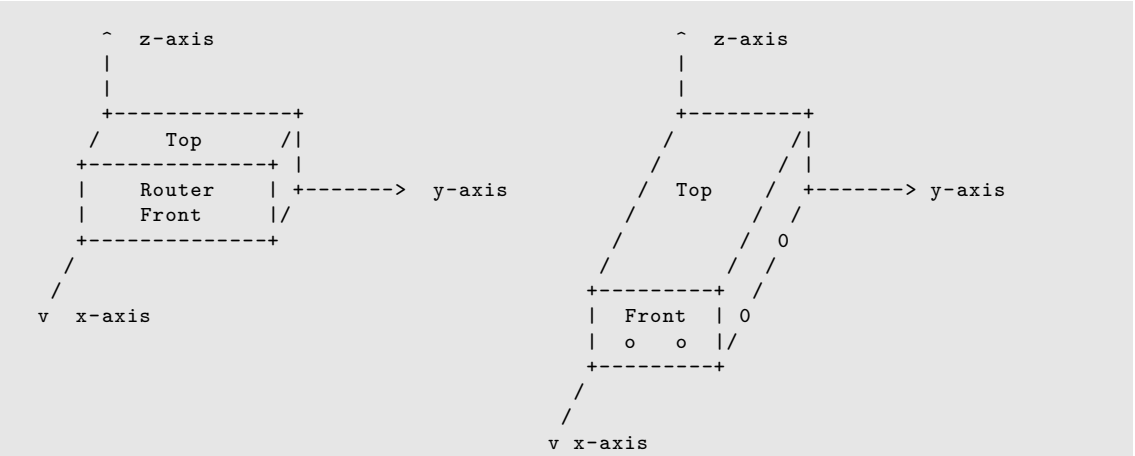
### 8.3.12 "imu" Subsystem

The `imu` subsystem configures Inertial Measurement Units to compensate Accelerometer and Gyroscope orientation, so their measurements are relative to the Mobile Router surrounding, such as a vehicle. It also configures some parameters for dead-reckoning inertial navigation, where applicable. It takes two optional subsections, `orientation` and `dead-reckoning` [v4.2.0]. The `orientation` subsection has the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `x` | integer | Rotation along x-axis, degrees | No |
| `y` | integer | Rotation along y-axis, degrees | No |
| `z` | integer | Rotation along z-axis, degrees | No |

The `orientation` defines correction parameters to map router orientation to vehicle/surrounding orientations, so that IMU measurements are provided relative to the vehicle, and also for dead-reckoning in routers with receivers that support it. IMU sensor orientation within the router is implicitly adjusted to be relative to the router chassis, so orientation adjustments are needed only to map IMU data to a coordinate system relative to the vehicle. All orientation correction axis rotation values are optional, omitted values implicitly define a rotation by zero degrees.

Without orientation correction, a router provides IMU data relative to the coordinate system shown below to the left, where the router front is the side showing the onway logo. The coordinate system for the vehicle is similar and shown to the right:




To compensate values to be relative to the vehicle, the router orientation is rotated around the axis specified; positive values apply rotation of values counter-clockwise in the right-handed coordinate system. Rotation is applied on the `x` -axis first, followed by the rotation along the `y` -axis, completed by the rotation of the `z` -axis.

For most installations, it is sufficient to give a rotation of `90`, `180` or `-90` degrees for the `z`-axis to compensate for router installation vs. vehicle orientation. For these common compensations, the following rotations along the `z`-axis can be used:

- Router front facing to vehicle back: `180`
- Router front facing to vehicle right (in driving direction): `-90`
- Router front facing to vehicle left (in driving direction): `90`
- Router front facing to vehicle front: No correction required

The `dead-reckoning` v4.2.0 subsection configures parameters for dead-reckoning inertial navigation, where supported by the GNSS receiver. It takes the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `dynamic-model` | string | One of `automotive`, `scooter` or `railway` | No |

The `dynamic-model` option selects a dynamic platform model that defines the expected vehicle dynamics to use for dead-reckoning sensor fusion. Not all models are supported on all dead-reckoning receivers. The default model is `automotive`.

Example:

```
imu:
  orientation:
    z: -90
  dead-reckoning:
    dynamic-model: automotive
```

### 8.3.13 "io" Subsystem

The `io` v4.3.0 subsystem specifies I/O related options.

#### state

The `state` section allows to configure the output state of I/O pins.

Each subsection of the `state` section configures the I/O module whose name is the subsection's name. In each of these subsections, a section for each pin can be defined which takes the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `state` | string | Requested state for the pin, `high` or `low` | Yes |

**Warning**

Setting the wrong pin to the wrong state can lead to short-circuits and potential equipment damages. Before configuring any pins, *always* check the router's electrical connection schema to avoid any damages.

**Note**

Pins that are not configured in this subsystem are set as inputs as a safe default state.

**Note**

When setting I/O pins as outputs in a conditional configuration, both condition cases should be covered using the `else` statement or the unconditional configuration. If a pin is only set as `high` or `low` in case a condition is fulfilled, this pin will be an input when the condition isn't fulfilled. This can lead to wrong signal state readouts by the connected device as the pin voltage level won't be tied to ground or the supply voltage.

Example for setting the pin `31` of the `g403-slot3` I/O module to `high`:

```
io:
  state:
    g403-slot3:
      "31":
        state: high
```

### 8.3.14 "`itxpt`" Subsystem

The `itxpt` subsystem enables service announcements compliant to the ITxPT standard used in public transport. The subsystem is configured per-interface, and each configuration accepts `services` by defining service subsections. Currently supported are the `inventory`, `gnsslocation` v3.9.2, `apc` v3.17.0 and `time` v3.18.0 services.

The mandatory `inventory` service takes the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `http` | integer | Port of HTTP/XML server | No |

If the `inventory` service `http` option is given, the inventory data is provided in XML format over HTTP at the specified TCP port, in addition to the MDNS TXT records provided implicitly.

The `gnsslocation` v3.9.2 service takes the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `port` | integer | Multicast port, defaults to `14005` | No |
| `group` | string | Multicast address, defaults to `239.255.42.21` | No v3.10.1 |

The `port` and `group` options define the port and multicast address where the location updates in XML format will be multicast.

The `apc` service acts as an APC *vehicle* application, discovering APC *door sensors* and collecting Automatic Passenger Counting figures on the associated network. The `apc` service has no options.

The `time` service advertises an independent (S)NTP server address via ITxPT, which might be internal or external to the router. This service alone does not provide any (S)NTP information, but may point ITxPT compatible devices to the location of an (S)NTP server, such as one provided by onway routers through the `ntp` subsystem. This internal NTP server would

have to be configured explicitly by adding a `ntp` section to the netconfig. This service takes the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `sntp-host` | string | Independent (S)NTP server IP address | Yes |

Example:

```
itxpt:
  lan0:
    services:
      inventory:
        http: 80
      gnsslocation:
        port: 1337
        group: 239.255.42.21
      apc: {}
      time:
        sntp-host: 10.10.1.1
```

### 8.3.15 "`link`" Subsystem

The `link` [v3.2.0] subsystem configures the router's interfaces. This includes setting Ethernet interface administrative state up or down, as well as configuring their link speeds, duplex modes and autonegotiation settings. The `link` [v3.11.0] subsystem also allows to set an alias name on an interface. The `link` subsystem allows to set Power over Ethernet properties on an interface if the router and the interface support PoE. Interfaces can be configured using the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `alias` | string | Interface alias name to assign | No |
| `mtu` | integer | Maximum Transmission Unit, defaults to `1500` | No |
| `up` | bool | Administrative interface state, defaults to `True` | No |
| `modes` | list[object] | List of Ethernet link modes to use | No |
| `poe` | object | Power over Ethernet properties to set | No |

The `modes` and the `up` option are only applied to Ethernet interfaces with a `lan` or `sfp` prefix. The `alias` option can be set on any interface.

The `mtu` [v4.0.0] sets the Maximum Transmission Unit, the minimum is `576` bytes, the maximum is `1500` bytes.

The `modes` option takes a number of Ethernet link modes that will be enabled for that interface on the router. The number of link modes given determines the autonegotiation behaviour. If a single link mode is given, the interface will be fixed to the speed and duplex settings of that link mode, and autonegotiation will be disabled. This should only be used with an Ethernet link peer that is configured to use the same link mode exclusively and without autonegotiation. If multiple link modes are given, they will all be enabled and advertised on the Ethernet interface, and autonegotiation will be enabled. If no link modes are given,

all link modes supported by the router will be enabled and advertised, and autonegotiation will be enabled. The link modes in `modes` each support the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `speed` | integer | Link speed, either `10`, `100` or `1000` Mbit/s | Yes |
| `half-duplex` | bool | Use half- instead of full-duplex mode, defaults to `False` | No |

For Gigabit Ethernet (1000BASE-T) and above, the only allowed duplex mode is full-duplex, and autonegotiation will always be enabled, as is mandated by the respective IEEE standards. Consult the router's datasheet to make sure that all Ethernet ports are capable of their required link speeds, as some ports may only support up to 100BASE-T.

The `poe` [v4.2.0] option takes currently one property, the `pse` flag. With this flag one can enable/disable the Power Sourcing Equipment for that interface.

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `pse` | bool | Power Sourcing Equipment, defaults to `True` | No |

**Note**

The `poe` feature is limited to routers with PoE support. Refer to the router manual for PoE configuration details.

Configuration example:

```
link:
  lan0:
    alias: myalias
    mtu: 1496
    up: True
    modes:
    - speed: 100
      half-duplex: True
    - speed: 1000
  lan2:
    poe:
      pse: false
```

### 8.3.16 "`lldp`" Subsystem

The `lldp` [v3.2.0] subsystem configures LLDP services to enable Link Layer Discover Protocol communication. The `lldp` can configure multiple instances, each running on a set of interfaces. Each instance uses a unique name and takes the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `interfaces` | list[string] | Interfaces to enable LLDP communication on | No |
| `snmp` | string | SNMP configuration this LLDP instance is accessible from | No |

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `hostname` | string | The LLDP instance uses this hostname as SysName | No |

The `interfaces` option is a list of interfaces where LLDPDU are sent and received. If the option is omitted or empty, the LLDP instance will use all physical interfaces.

If the `snmp` option is given, it must refer to a `snmp` subsystem agent configuration by its network interface name. The referenced SNMP instance MIB is extended by the LLDP information provided by this LLDP instance, that is, for itself and any LLDP neighbour discovered on the configured interfaces. The `hostname` [v3.2.1] option allows to set a custom hostname in the LLDP environment. The hostname is seen as `SysName` LLDP TLV. Example:

```
lldp:
  myconfig:
    interfaces:
    - lan2
    - lan3
    snmp: vrf10
    hostname: hostname.example.com
```

### 8.3.17 "`logging`" Subsystem

The `logging` subsystem configures output sinks for syslog messages from the Mobile Router, and provides filtering capabilities to produce log output with an appropriate verbosity. The `logging` subsystem takes sections with arbitrary logging configuration names, and each accepts these options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `output` | object | Log output sink configuration | Yes |
| `levels` | object | Log level filtering configuration | Yes |

The `levels` section takes key/value pairs, where the key is the *syslog identifier* used for logging, usually the service name. The value is an integer syslog log level to log up to, a number from `0` to `7`. `0` logs only the most urgent emergency log messages, `7` includes all log messages, including debugging logs. The special syslog identifier `default` defines the log level to use for messages that carry an identifier not matching any explicit setting. If no `default` is provided, messages are ignored unless a specific subsystem log level tells otherwise.

Under the `output` section, subsections named `file` and `rsyslog` can be specified. The `file` section logs to a local file on the Mobile Router, and takes these options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `path` | string | Filename to write to | Yes |
| `size` | integer | Size limit of file before rotating it, in bytes | No |
| `keep` | integer | Number of rotated files to keep | No |

If the `size` and `keep` options are omitted, the log file is not rotated and keeps growing indefinitely. The `rsyslog` `output` sink logs to a remote syslog server using the UDP transport mechanism, and is configured with:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `server` | string | Syslog server hostname or IP address | Yes |
| `port` | integer | Syslog server UDP port | No |
| `hostname` | string | Local system hostname to include in messages | No |
| `ifname` | string | Interface name to bind client socket to | No |

The `server` option takes domain names to resolve, or an IPv4/IPv6 address. The `port` option defaults to the standard syslog port `514`. If the `hostname` option is omitted, the system hostname is used. Specifying an `ifname` allows the syslog client socket to bind to an interface or VRF domain. Example:

```
logging:
  local-debug:
    output:
      file:
        path: /data/log/debug.log
        size: 256000
        keep: 5
    levels:
      default: 7
  remote:
    output:
      rsyslog:
        server: syslog.example.org
        ifname: vrf0
    levels:
      hostapd: 6
      nacho: 5
```

### 8.3.18 "modem" Subsystem

The `modem` subsystem configures different aspects of cellular modem operation, namely:

- Foreign network roaming policies in the `roaming` section
- SIM slot to modem assignment in the `slots` section
- Custom APN configurations in the `apn` section
- Network configurations in the `net` section
- SIM PIN configurations in the `pin` section

#### Roaming Policy

When a modem is connected to the home network indicated by the SIM card, PDP and IP sessions are implicitly established. On a foreign network, such connections may often result in additional costs charged by the operator, hence they can be globally or selectively enabled and disabled. The `roaming` section in the `modem` subsystem takes the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `default` | bool | Default roaming policy | No |
| `allow` | object | Roaming allow rule-set overriding `default` | No |
| `deny` | object | Roaming deny rule-set overriding `default` | No |

The `default` option defines the roaming policy to use if no explicit `allow` or `deny` rule matches. If the `default` option itself is omitted, it defaults to `False`, prohibiting roaming.

**Note**

While roaming is not allowed if not explicitly specified in the configuration, a Mobile Router has `default` set to `True` in its *factory configuration*. This allows a device to bootstrap its configuration on foreign networks.

The `allow` and `deny` rule-sets define explicit rules to apply for specific modems by name or by the IMSI prefix given by the inserted SIM card. Each rule uses a match definition as a list name, which is either an IMSI prefix of the SIM card used, or the full interface name of the modem. Under that list, Mobile Country Codes (MCCs) match the country of the network to (refuse to) roam to.

If one or more rules match with the IMSI prefix or the modem interface and a listed foreign network MCC, the rule with the longest IMSI prefix match is selected; IMSI prefix matches have higher priority over interface name matches. The best matching rule defines the roaming policy applied.

```
modem:
  roaming:
    default: False
    allow:
      mbm0:
      - 262
      - 234
    deny:
      "295":
      - 234
```

In the example above, roaming is denied by default. On `mbm0`, roaming is allowed in Germany (262) and UK (234) networks, unless a SIM card from Liechtenstein is used, in which roaming in the UK is denied.

#### SIM Slot Configuration

Some Mobile Routers have more SIM slots than physically available modems, and support the dynamic assignment of SIM slots to modems. The `slots` section takes lists named after the modem to assign slots to, containing one or more SIM slots to potentially assign to a modem.

Slots are numbered starting from 1. If a modem has multiple slots assigned, slots with an inserted SIM are used. If IP connectivity with a SIM in a slot fails, the system tries alternative slots that are configured for the modem. By default, SIM slots are assigned sequentially, that its `mbm0` is assigned slot `1`, `mbm1` to slot `2`, etc. Any slot must be assigned to a single modem, only. Some router models may have further restrictions on slot assignment.

```
modem:
  slots:
    mbm0:
    - 1
    mbm1:
    - 2
    - 3
```

In the example above, the first SIM slot is assigned to `mbm0`, whereas `mbm1` alternatively tries the secondary and third slot until it finds a working SIM.

#### Custom APN Settings

The `modem` subsystem has a built-in database of public APNs for common network operators to automatically use for establishing IP connectivity. To connect to private APNs, explicit APN configuration is required in the `apn` section. APN configurations can be defined for a specific modem interface or for a IMSI prefix of the SIM inserted in any modem-assigned SIM slot. The IMSI prefix is the concatenation of the MCC and the MNC of the SIM card operator. Each APN configuration takes the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `apn` | string | APN name | Yes |
| `username` | string | APN authentication username | No |
| `password` | string | APN authentication password | No |
| `auth` | string | APN authentication type: `pap` or `chap` | No |
| `mtu` | integer | Custom MTU to use on network | No |
| `routes` | list[string] | List of CIDR network routes to use for APN | No |

The used authentication method is automatically selected, unless an explicit authentication method is requested using the `auth` v3.1.1 option. The `mtu` v3.17.1 option sets the Maximum Transmission Unit (MTU) to use on the modem for this APN, defaulting to 1428 bytes. If the `routes` option is given, the modem forwards traffic to the listed destinations only instead of routing all traffic to any destination. This is useful for APNs that provide access to internal networks only, but not to the Internet. Example:

```
modem:
  apn:
    mbm0:
      apn: foo
    "22801":
      apn: bar
      username: baz
      password: s3CuRe
      mtu: 1320
      routes:
      - 10.0.0.0/8
```

#### Network Configuration

The `net` [v3.4.0] section controls the IP network configuration of a modem. The configuration entries are separate for each modem and to be listed under the respective modem interface name. Options include the `use-dns` flag, which can be set to `True` to learn DNS servers from the mobile network operator. This enables the router to forward DNS requests over the correct modem uplink when dealing with internal networks.

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `use-dns` | bool | Learn DNS servers from mobile operator | No |

```
modem:
  net:
    mbm0:
      use-dns: True
```

#### SIM PIN Configuration

SIM cards can optionally be protected by PINs. It is usually recommended to use SIM cards with PIN protection disabled, as it is a common case for configuration errors and prevents a router from bootstrapping over modem uplinks using a factory configuration.

The `pin` [v3.6.0] section in the `modem` subsystem defines the PINs to use if a SIM requires PIN verification. A SIM can be identified either by its `iccid` or by the `slot` number it is inserted to. Specifying PINs by ICCID is the recommended method, as it uniquely identifies a SIM card, regardless of its physical location.

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `iccid` | object | PINs defined for specific SIM cards by ICCIDs | No |
| `slot` | object | PINs defined for SIM cards by SIM slot | No |

The `iccid` section takes mappings from ICCID SIM card numbers to PINs, regardless of its physical location, whereas the `slot` section takes mappings from SIM slot numbers to PINs. Matching `iccid` entries have preference over matching `slot` entries if a SIM card has matches for both types of entries. PINs are numerical with a length of four to eight digits, but must be specified explicitly as strings to support PINs with leading zeroes. Example:

```
modem:
  pin:
    iccid:
      89410118124724140197: "012345"
    slot:
      1: "1455"
```

### 8.3.19 "`multicast`" Subsystem

The `multicast` subsystem configures multicast protocol support and forwarding mechanisms. It currently supports:

- Multicast reflection in the `reflect` [v3.11.0] section

#### Multicast reflection

Multicast reflection is a mechanism to exchange UDP multicast packets between two different locally attached layer 2 networks. It forwards packets without any manipulation to the IP/UDP layer, the source address and the IP TTL is preserved. Reflection explicitly violates multicast routing standards, and is not restricted to multicast groups that use a routable address or packets with an IP TTL greater than 1. It is useful to proxy services using non-routable multicast groups, such as mDNS and ITxPT.

The `reflect` configuration takes a list of configuration objects, each taking the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `ifname` | string | Input interface to listen for multicasts | Yes |
| `group` | string | Multicast group to forward | Yes |
| `output` | list[string] | List of interfaces to forward multicast to | Yes |

Multicast reflection receives UDP multicast packets on `ifname` for the multicast address `group`. Any received packet is reflected to all defined network interfaces in `output`. The combination of `ifname` and `group` must be unique, but multiple configurations for the same interface can exist. Example for bidirectional mDNS reflection:

```
multicast:
  reflect:
  - ifname: lan0
    group: 224.0.0.251
    output:
    - lan1
  - ifname: lan1
    group: 224.0.0.251
    output:
    - lan0
```

### 8.3.20 "nac" Subsystem

The `nac` subsystem configures Network Access Control policies. NAC is enforced using 802.1x authentication with EAP or with MAC Authentication Bypass (MAB) for legacy clients to just authenticate the client's MAC address. Both mechanisms rely on a RADIUS backend for authentication. The `nac` subsystem takes arbitrarily named configuration sections, each with the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `method` | string | Limit authentication method | No |
| `ports` | list[string] | Access port interface names to run NAC on | Yes |
| `policy-map` | object | Mapping of policy to interface name suffixes | Yes |
| `radius` | object | RADIUS server configuration section | Yes |

NAC port elevation is implemented by adding MACVLAN interfaces onto an access port network interface (the *parent*). MACVLANs operate in source mode, so an authenticated client is allowed to one of the MACVLAN interfaces by adding its MAC address to the interface. A

client that has not (yet) been elevated is in the *unauthenticated* state and stays on the Access Port. Network connectivity in the unauthenticated state (on the Access Port) or in an elevated state (on a MACVLAN interface) is not directly managed by Network Access Control; instead, the MACVLAN interfaces can be enslaved to VRF or bridge interfaces using the `vrf` and `bridge` subsystems, respectively. Parent Access Port interfaces currently do not support bridging, but can be isolated using VRFs.

The `method` [v3.3.0] limits the authentication method to `mab` or `eap`. If it is omitted, then both authentication mechanisms are active.

NAC clients get authenticated using a RADIUS server, the `radius` section takes the following sub-options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `nasid` | string | NAS identifier | Yes |
| `client-addr` | string | IP address used for RADIUS client | No |
| `client-dev` | string | Interface name to bind RADIUS socket to | No |
| `degrade` | integer | Non-responsive server degradation, in s | No |
| `message-auth` | bool | Require Message-Authenticator | No |
| `servers` | list[object] | RADIUS servers | Yes |

A `client-addr` forces a specific address to use for RADIUS communication. A VRF for RADIUS communication can be directly or indirectly selected using the `client-dev` option, the VRF can be different from the VRFs used by Access Ports. The `servers` option is a list of servers to try to authenticate clients against, each taking the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `server` | string | RADIUS server IP address | Yes |
| `secret` | string | RADIUS secret | Yes |

Multiple RADIUS servers can be specified, the servers are used in the priority of their appearance in the configuration. A server that is non-responsive gets degraded and prioritized below all other responding servers. The `degrade` [v3.8.3] option, defaulting to 180s, defines how long such a server stays in degraded state before it gets retried.

If the `message-auth` [v3.16.0] option is defined and set to `True`, the RADIUS client requires the Message-Authenticator attribute to be present and valid in MAB RADIUS response messages. Enabling this behavior fixes the Blast-RADIUS vulnerability in the RADIUS protocol, but requires RADIUS servers to include the attribute in non-EAP RADIUS responses. It affects MAB authentication only, as EAP authentication always requires the Message-Authenticator attribute. Setting the option is recommended for security reasons, but defaults to `False` for compatibility reasons.

While 802.1x can authenticate clients securely with EAP over RADIUS, MAB authentication verifies the MAC address only, and encodes the MAC address in the following RADIUS attributes when sent to the backend:

- `Username`: Lower case hex-encoded MAC address, `001122aabbcc`
- `User-Password`: As in Username, but using RADIUS `User-Password` encryption
- `Calling-Station-Id`: Upper case hex-encoded with dashes, `00-11-22-AA-BB-CC`

The RADIUS AAA backend selects an authentication policy by responding with an *Access-Accept* or an *Access-Reject*. The configurations `policy-map` describes the mapping of the AAA policy to MACVLAN interface name suffixes. A MACVLAN interface is created for any defined suffix by appending the suffix to any of the Access Port interface names given in the `ports` option, separated by a dot. The `policy-map` takes one or more entries for the following policy types:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `accept` | string | Default accept policy interface suffix, used for *Access-Accept* verdicts without a more specific match | No |
| `reject` | string | Default reject policy interface suffix, used for *Access-Reject* verdicts | No |
| `vlanid:X` | string | VLAN ID to interface suffix map, used for *Access-Accept* verdicts containing a VLAN ID `X` | No |

To standardize naming, the suffix for `accept` or `vlanid` policies must be `accX`, where `X` is an optional number, resulting in MACVLAN interface names such as `lan0.acc0`. For the reject policy, the suffix must match `rej`, creating a MACVLAN interface name such as `lan1.rej`.

If the `reject` policy is missing, a rejected client stays in the *unauthenticated* state on the parent Access Port interface, so does it when the RADIUS server is unreachable. For clients using 802.1x, a `reject` policy may not allow a client to access the associated interface, as a client usually blocks access if EAP authentication fails. Some clients support falling back to *unauthenticated* access, and then can get assigned to the reject policy interface if MAB authentication is rejected.

If VLAN information attributes are included in an *Access-Accept*, the policy matches to a `vlanid` policy with that VLAN identifier. Note that VLAN identifiers from RADIUS are used for the sole purpose of matching a policy; It defines the target Layer 2 network to attach, regardless of a potential VLAN encapsulation used in or on such a network. VLAN identifiers in RADIUS responses must be encoded according to RFC 3580, using:

- `Tunnel-Type`: VLAN (`13`)
- `Tunnel-Medium-Type`: 802 (`6`)
- `Tunnel-Private-Group-ID`: String encoded VLAN number

If no VLAN policy matches, the default `accept` policy is applied instead. If no matching `vlanid` policy map entry exists and no `accept` policy is defined, RADIUS *Access-Accept* verdicts keep the client in the *unauthenticated* state. Example:

```
nac:
  myconfig:
    method: mab
    ports:
    - lan2
    - lan3
    policy-map:
      accept: acc
      reject: rej
      vlanid:7: acc1
```

```
  radius:
    nasid: mynas
    client-addr: 10.0.0.1
    client-dev: vrf10
    servers:
    - server: 10.9.0.22
      secret: foo/bar
```

### 8.3.21 "network-tests" Subsystem

The `network-tests` subsystem configures network tests to perform, generating network traffic patterns defined by *schemes*. Configured tests are constantly run against a target server, which must run the appropriate service to terminate the tests. Test results with detailed statistics can be collected using the `network-tests` telemetry source.

#### Client Configuration

The `network-tests` subsystem `client` [v3.9.0] section accepts sections for arbitrary test names to run as client, containing the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `scheme` | string | Traffic scheme name | Yes |
| `connect` | string | Server IP address to test against | Yes |
| `bind` | string | Client IP address to bind to | No |
| `options` | object | Options to use for traffic scheme | Yes |

The different traffic schemes accept different options. Only some options are accepted for each scheme from the following set of options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `interval` | integer | Interval to run test in, in ms | No |
| `up-rate` | integer | Client upload rate, in kbit/s | No |
| `down-rate` | integer | Client download rate, in kbit/s | No |
| `pkt-size` | integer | Size of UDP packet payload, in bytes | No |
| `count` | integer | Number of actions to perform each interval | No |
| `buf-size` | integer | Buffer size for stream caching, in kB | No |
| `file-size` | integer | File size for fetches, in kB | No |
| `ifname` | string | Interface name to send packets from | No |

Details about schemes, their configuration and their provided measurement results is out of the scope of this manual. Example:

```
network-tests:
  client:
    ping:
      scheme: icmp
      connect: 10.0.0.1
      options:
        interval: 500
        pkt-size: 1000
```

```
        ifname: lan0
    http:
      scheme: fetch
      connect: 10.0.0.1
      bind: 10.0.0.2
      options:
        interval: 1000
        file-size: 512
```

#### Server Configuration

The `network-tests` subsystem `server` [v3.9.0] list accepts server instances that clients can use for terminating tests. Each server object takes the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `ifname` | string | Interface or VRF to bind server to | No |

Multiple server instances can be created by using different `ifname` options. If the `ifname` option is omitted in an instance, the server listens on all interfaces in the default VRF. Example:

```
network-tests:
  server:
  - ifname: vrf10
```

### 8.3.22 "ntp" Subsystem

The `ntp` subsystem configures NTP server instances to provide accurate time information to local clients. Mobile Routers itself synchronize their system time by other means using different time sources, but the system time can be provided to local clients using the Network Time Protocol via this subsystem. The `ntp` subsystem is a list of configurations, each taking the following option:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `ifname` | string | Interface name to bind NTP server to | Yes |

The `ifname` option defines the interface to bind the NTP server instance to, which can be used to select a VRF domain. The NTP server listens on UDP port `123`. Example:

```
ntp:
- ifname: lan0
- ifname: vrf10
```

### 8.3.23 "radv" Subsystem

The `radv` subsystem configures IPv6 router advertisements to announce to local networks. It takes per-interface subsections, each with the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `router` | bool | Act as default router | No |

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `nameservers` | list[string] | List of Nameservers to announce | No |
| `prefixes` | list[object] | List of IPv6 prefixes to announce | No |

If the `router` option is present and `True`, the Mobile Router announces itself as a default router, so clients may use it as a next-hop for their IPv6 traffic. The `nameservers` list announces DNS servers to clients using RFC 8106 options. The `prefixes` section takes subsections for prefixes in CIDR notation to announce on the configured interface. Each prefix nests the following option:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `slaac` | bool | Allow StateLess Address AutoConfiguration | No |

If the `slaac` option is present and `True`, a client may generate an address under that prefix using SLAAC, as defined in RFC 4862. Example:

```
radv:
  lan0:
    router: True
    nameservers:
    - fec0:1111::10
    - fec0:2222::10
    prefixes:
      fec0:1234::/64:
        slaac: True
```

### 8.3.24 "route" Subsystem

The `route` subsystem installs and maintains statically configured network routes for IPv4 and IPv6. The `route` property is a list with objects, each representing a single route, with the following properties:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `dst` | string | CIDR notation network of route destination | Yes |
| `via` | string | Next-hop gateway to route over | No |
| `ifname` | string | Output interface to route over | No |
| `metric` | integer | Route priority | No |
| `table` | integer | Table id to install route into | No |

`metric` is an arbitrary number, and for routes with equal `dst` subnets, the route with the lower metric is preferred. If a `table` is given, it may refer to the table of a VRF in a `vrf` subsystem definition to use a route for that VRF. Either a `via` nexthop gateway or an output `ifname` [v3.17.0] must be given. Example:

```
route:
- dst: 10.0.0.0/16
  via: 10.0.0.1
  ifname: lan0
  metric: 100
- dst: ::/0
```

```
via: fc00::1
table: 10
```

The next-hop must be reachable for the system, for example by being locally attached using the prefix given in an address entry defined in the `addr` subsystem. Inter-family routes are not supported.

### 8.3.25 "`service-check`" Subsystem

The `service-check` [v3.13.0] subsystem provides the ability to monitor external devices and services through different types of sensors. Results are reported back through the telemetry service. The `service-check` section takes arbitrarily named group subsections[v4.5.0] to help organize different service checks and reference them under one name. Each named group subsection accepts the following sensor types as subsections. Each sensor can be configured with multiple targets to check:

- `daemon` [v4.5.0]: Checks CarlOS system daemon health by querying them. This sensor expects the name of the daemon to monitor. The supported daemons as well as their behaviour is explained below.
- `http`: Checks web servers by sending them HTTP(S) GET requests and checking the returned HTTP status code. On connection problems or other general errors, an error string is reported instead. This sensor expects service URLs to check as per-target section names.
- `icmp`: Checks services using ICMP echo requests (ping) and reporting the average round-trip time (RTT) and loss rate over the last couple of pings. This sensor expects IPv4/6 addresses to check as per-target section names.
- `snmp`: Checks services using SNMP GetRequests to report the current value of multiple OIDs. This sensor expects IPv4/6 addresses of SNMP agents to check as per-target section names. For each target, an arbitrary number of OIDs to check can be provided, each with a optional time interval which overrides the global interval of the service target.

The `daemon` service target can be used to monitor following daemons:

- `image`: Health of the `image` service. Fails if the system has booted from the fallback partition.
- `vpn`: Health of the `vpn` service. Fails if any of the configured VPN tunnels are down due to permanent errors such as missing credentials, or configuration mismatches. For non-permanent errors like network connectivity loss, the service check reports a success.

The `daemon` service target object accept the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `interval` | integer | Interval to check service, in seconds | Yes |

The `http` and `icmp` services targets objects accept the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `interval` | integer | Interval to check service, in seconds | Yes |
| `ifname` | string | Force network interface over which to check | No |
| `assertion` | object | Conditions to assert for the service check | No |

The `assertion` [v4.5.0] option allows to define custom conditions for the service check to be considered successful. If no assertion is given, checks fall back to sensor-specific default conditions.

For the `http` sensor, the default condition is that the connection succeeded and the returned HTTP status code is between 200 and 299, inclusive. Custom assertions support the following conditions:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `code` | integer | Expected HTTP status code between 100 and 599 | Yes |

For the `icmp` sensor, the default condition is that the ICMP packet loss rate is zero percent, the RTT (round-trip time) is not considered. Custom assertions support the following conditions:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `max-loss-rate` | integer | Maximum packet loss rate in percent | No |
| `max-rtt-average` | integer | Maximum average RTT in milliseconds | No |

To reduce the jitter in the results, which could cause false alarms, the ICMP service checker employs an exponential moving average filter smoothing the results. On service startup, this filter needs twelve measurement intervals to reach a saturated state for correct operation. Therefore, results for the ICMP service checks are delayed and only reported after the filter is ready.

The `snmp` service target accepts the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `oids` | list[object] | List of OIDs to check | Yes |
| `interval` | integer | Interval to check service, in seconds | Yes |
| `ifname` | string | Force network interface over which to check | No |
| `community` | string | Community string to use with target agent, defaults to `public` | No |

For each OID to check, the following options are accepted:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `oid` | string | OID to check | Yes |
| `interval` | integer | Interval to check OID, in seconds. Defaults to the service target interval. | No |
| `assertion` | object | Conditions to assert for the OID check | No |

The `assertion` [v4.5.0] option allows to define custom conditions per OID for the service check to be considered successful. If no assertion is given, any SNMP response that's not an error is considered successful. Custom assertions support the following conditions:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `value` | string or integer | Expected value of the OID | Yes |

**Note**

It's not recommended to configure identical service checks in multiple groups simultaneously, since the telemetry results do not include the group name, and thus cannot be distinguished.

Example:

```
service-check:
  group01:
    icmp:
      1.1.1.1:
        interval: 10
        ifname: lan0
        assertion:
          max-loss-rate: 10
          max-rtt-average: 100
    http:
      "https://example.com":
        interval: 20
        assertion:
          code: 200
    snmp:
      10.0.0.1:
        oids:
        - oid: 1.3.6.1.2.1.1.6.0
          interval: 5
          assertion:
            value: "telephone closet, 3rd floor"
        - oid: 1.3.6.1.2.1.1.7.0
        interval: 10
        community: public
    daemon:
      vpn:
        interval: 15
```

### 8.3.26 "snmp" Subsystem

The `snmp` subsystem configures SNMP services to interface with Mobile Routers using the Simple Network Management Protocol. The `agent` section in this subsystem configures SNMP agents, which can be accessed read-only to monitor Mobile Router systems. The `agent` section takes configurations per network interface, each with the following option:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `hostname` | string | SNMP `SysName` to report | No |
| `users` | list[object] | User definitions for SNMP access | Yes |

Each user entry in the `users` list further takes the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `user` | string | SNMPv3 user name | Yes |
| `password` | string | SNMPv3 user password | No |

Currently, only SNMPv3 is supported. Any user has read-only permissions, and privacy is currently not supported. If no `password` is given, no authentication is performed (RFC 3414 *NoAuthNoPriv*). If a `password` is provided, it must be at least 8 characters long, and enables authentication (*AuthNoPriv*) using the SHA-256 authentication protocol (RFC 7860 *usmHMAC192SHA256AuthProtocol*). The `hostname` v3.3.0 option allows to set a custom hostname, seen as `SysName` in the SNMP System Group. Example:

```
snmp:
  agent:
    lan0:
      hostname: hostname-lan0.example.com
      users:
      - user: foo
        password: a-(not-so-)strong-password
    vrf10:
      hostname: hostname-vrf10.example.com
      users:
      - user: open
```

### 8.3.27 "system" Subsystem

The `system` v3.10.0 subsystem configures system settings.

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `hostname` | string | Hostname | No |

The `hostname` option allows configuring a global hostname for the router. If omitted, it defaults to the router serial number. The global hostname is used as default hostname in `lldp`, `snmp` and `logging` subsystems if they do not define a specific hostname explicitly.

```
system:
  hostname: my-hostname.example.com
```

### 8.3.28 "users" Subsystem

The `users` subsystem configures Mobile Router system users. Under the `local` section, it accepts local Unix user accounts with username/password that can be used for Serial Console logins or logins via SSH. Each username defines its own section, and under it the following option:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `password` | string | Hashed user password string | Yes |

The password for a local user is provided in hashed form using the *crypt*[11] format. The hash algorithms `$1$` (MD5), `$5$` (SHA-256) and `$6$` (SHA-512) are supported. Hashed passwords can be generated using the `openssl passwd` utility with the `-1`, `-5` or `-6` options. All herein specified users can use the `sudo` utility to gain root privileges. Example:

```
users:
  local:
    tester:
      password: $1$NQlpkjWb$x9mognHJ5Y4XPFomHqLmh1
```

**Note**

Logging in to onway Mobile Routers can be useful for troubleshooting. Configuration changes applied on the console are not persistent, though, and are usually lost during a reboot. Router configuration management shall be done through the Mixer configuration tool, only.

### 8.3.29 "vlan" Subsystem

The `vlan` subsystem installs 802.11q VLAN interfaces on top of physical untagged Ethernet parent interfaces. VLAN interfaces filter and remove a matching VLAN tag on frames received on the parent interface, and when sending over the VLAN interface add the VLAN tag to frames before forwarding them over the physical parent interface. VLAN interfaces can be be associated to VRF domains or associated to Layer 2 bridges, independent of the parent interface or other VLAN interfaces on the same parent.

The `vlan` section takes subsections for VLAN interfaces to create and maintain. Each interface section takes the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `vlanid` | integer | 802.11q VLAN identifier for interface | Yes |
| `parent` | string | Parent Ethernet interface to operate on | Yes |

The `parent` must be an Ethernet interface, either a physical interface, a bridge interface defined in the `bridge` subsystem, or an interface defined by the `container` or `vm` subsystems. To make interface names intuitive, VLAN interface names are currently restricted to the form `parent.vlanid`, where `parent` and `vlanid` shall match the options used for that VLAN interface. Example:

```
vlan:
  lan0.22:
    vlanid: 22
    parent: lan0
  br0.7:
    vlanid: 7
    parent: br0
```

[11] https://en.wikipedia.org/wiki/Crypt_(C)

### 8.3.30 "vm" Subsystem

The `vm` [v3.8.0] subsystem configures virtual machines to run in the Mobile Router Hypervisor. Details about the virtualization solution and the recommended storage strategy can be found in the separate documentation [12]. The subsystem takes arbitrary named subsections with virtual machine names, each accepting the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `mem` | integer | Guest VM memory size, in MB | Yes |
| `cores` | integer | Guest VM CPU core count | No |
| `base` | object | VM read-only base image | Yes |
| `overlays` | list[object] | Copy-on-Write overlay image stack | No |
| `disks` | list[object] | Plain extra disks to attach | No |
| `network` | list[object] | Network interfaces between VM and host | No |
| `can` | list[object] | CAN interfaces between VM and host | No |
| `vnc` | list[object] | VNC server listen ports | No |
| `dmi` | list | DMI/SMBIOS variables to set for VM | No |

The `mem` option defines the RAM size the guest VM has available, in Megabytes. The `cores` options the number of virtual CPU cores, defaulting to `1` if omitted.

The `base` section defines the mandatory base disk image a virtual machine boots from. This image is read-only and is never written to, and takes these options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `location` | string | Base image storage location | Yes |
| `image` | string | RAW format base image filename | Yes |

The `location` option selects the storage location containing images, which can be a filesystem bundle or removable storage. The `image` defines the filename of the image to use as base image.

The list of `overlays` defines Copy-on-Write overlay images to stack above the base image, with the first image defined being the top-most image of the stack that is written to.

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `location` | string | Overlay image storage location | Yes |
| `image` | string | Overlay image filename | Yes |
| `purge` | string | Overlay purge mode, `on-stop` or `on-change` | No |

The `image` defines the filename of a disk image overlay in the specified writeable storage `location`, usually an internal disk. Non-existing images get created implicitly. If an `on-stop` `purge` mode is defined for an overlay, the overlay is cleared whenever a VM stops, discarding any changes done. Only the top overlay layer(s) can use `on-stop` purging. The default `on-change` purge mode purges an overlay only when the used base image has

[12] `https://support.onway.ch/onway-router/pubdoc/latest/virtual-machines.html`

changed. If the list of `overlays` is omitted or empty, changes to the base images are stored in host memory with limited capacity, and discarded if a VM stops.

The list of `disks` defines additional plain extra disk drives to pass to the VM:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `location` | string | Disk image storage location | Yes |
| `image` | string | Disk image filename | Yes |
| `size` | integer | Size of disk, in MB | Yes |

**Note**

The disks have to be partitioned and formatted by the VM operating system.

The `image` defines the filename of a RAW disk image in the specified writeable storage `location`. Existing images get resized to the specified `size`, given in Megabytes, non-existing images get created implicitly.

The `network` option allows configuring one or more TAP devices, virtual Ethernet devices between the VM and the host:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `ifname` | string | Host side interface name | Yes |
| `mac` | string | MAC address of interface, hex-encoded | No |

The `ifname` defines the host side interface name, whereas the VM side interface naming is up to the virtual machine operating system. On the host side, the interface can be used like an ordinary Ethernet interface, that is, associated to a bridge or a VRF. If the `mac` option is omitted, a MAC address will be generated deterministically based on the interface name.

The `can` v4.1.0 option allows configuring one or more CAN devices to exchange frames between a virtual CAN interface on the host and an emulated CAN device in the VM. Listing the host side CAN interface in the `can` subsystem `forward` section allows the virtual machine to receive frames from the associated CAN hardware. The `can` option takes a list of CAN interface objects, each with the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `ifname` | string | Host side interface name | Yes |

The `vnc` option takes a list of VNC server listening ports:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `port` | integer | TCP port number to bind VNC service to | Yes |
| `ifname` | string | Interface to bind VNC service to | No |

The `port` defines a TCP port VNC clients can connect to to get access to the guest VM primary display. If an `ifname` is given, the socket is bound to the specified interface or VRF. If omitted, the socket accepts connections on any interface in the default VRF.

Under the `dmi` [v3.10.0] option, DMI/SMBIOS variables can be defined for the guest VM, allowing per-instance variables to be passed to the guest. The following suboptions for DMI variables are defined:

- `bios-vendor`
- `bios-release-date`
- `bios-version`
- `system-manufacturer`
- `system-product-name`
- `system-version`
- `system-serial-number`
- `baseboard-manufacturer`
- `baseboard-product-name`
- `baseboard-version`
- `baseboard-serial-number`
- `baseboard-asset-tag`
- `chassis-manufacturer`
- `chassis-version`
- `chassis-serial-number`
- `chassis-asset-tag`, all taking arbitrary strings,
- `system-uuid` taking a string in UUID format, and
- `oem` being a list with arbitrary strings.

The following example creates a VM named `mediaportal` using a base image from a bundle with the same name. A volatile overlay stores disk changes while the VM is running, and an extra disk permanently stores media files. The VM is integrated to the host network via the host-side `vmtap0` interface and can be managed via VNC over an IPsec tunnel.

```
vm:
  mediaportal:
    mem: 2084
    cores: 2
    base:
      location: BUNDLE:mediaportal
      image: mediaportal.raw
    overlays:
    - location: SSD:1
      image: mediaportal.qcow2
      purge: on-stop
    disks:
    - location: SSD:1
      image: media.raw
      size: 16384
    network:
    - ifname: vmtap0
    can:
    - ifname: vmcan0
    vnc:
    - ifname: xfrm0
      port: 5900
```

```
  dmi:
    system-serial-number: ds=nocloud-net;s=http://10.0.0.1/meta
    system-uuid: dd75a06a-26f9-11ee-be56-0242ac120002
    oem:
    - DEBUG
    - DNS=portal.example.com
```

### 8.3.31 "voltage" Subsystem

The `voltage` subsystem configures voltage monitoring and actions to trigger upon voltage changes. Its primary use case is to monitor the vehicle ignition voltage and trigger system shutdown a few minutes after the vehicle engine has been turned off. This ensures that the router can provide its service for short vehicle stops, but that it does not drain the vehicle battery if the vehicle is stopped for a longer duration. The `voltage` subsystem takes a `settings` subsection with currently the following option:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `follow-up` | integer | Follow-up time before shutting down, in s | No |

If no explicit `follow-up` duration is defined, the default of `600` is used, resulting in a system shutdown 10 minutes after the ignition voltage drops below a threshold. The router is restarted once the ignition voltage recovers. Example:

```
voltage:
  settings:
    follow-up: 300
```

### 8.3.32 "vpn" Subsystem

The `vpn` subsystem configures VPN tunnels to establish to backend VPN gateway systems, along with associated parameters. The onway VPN solution builds upon standards-compliant IPsec and certificate based authentication using the IKEv2 key exchange protocol, but extends it with comprehensive multipath load-balancing capabilities, traffic separation, latency based QoS and cluster functionality. The `vpn` subsystem is configured through the following subsections:

- `tunnels` takes the main IPsec tunnel configurations
- `bypass` lists networks for traffic to exclude from tunnels
- `latencies` defines per-uplink target latencies to shape to
- `qos` defines traffic classes and their bandwidth reservation
- `cluster` combines multiple onway routers to act as a redundant gateway

More details about the capabilities provided by the onway Mobile Router VPN technology is available in a separate documentation[13].

[13] `https://support.onway.ch/onway-router/pubdoc/latest/uplink-management.html`

#### Tunnel Configuration

Under the `tunnels` subsection, the `vpn` subsystem takes a set of arbitrary named tunnel sections. Each takes the following list of options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `gateway` | string | VPN gateway hostname to connect to | Yes |
| `identity` | string | Local IKE identity to authenticate with | Yes |
| `xfrmi` | string | Name of XFRM interface to create for tunnel | Yes |
| `local-ts` | list[string] | Local CIDR subnets to send over tunnel | No |
| `remote-ts` | list[string] | Remote CIDR subnets to send over tunnel | No |
| `probe` | string | Latency probe server IP address | No |
| `uplinks` | object | Underlay uplinks to use for tunnel | Yes |

The `gateway` option defines the DNS name of the VPN gateway to initiate tunnels to, which at the same time is the IKE identity expected from the server. The server certificate must contain this identity as subjectAltName to establish trust. The `identity` defines the IKE identity used by the Mobile Router, and a certificate must be available on the router with such a subjectAltName.

The `xfrmi` option defines the name of a tunnel interface, which must match a `xfrm` prefix followed by an arbitrary number. Traffic sent to this interface gets encapsulated by the tunnel, and traffic decapsulated from the tunnel is received on this interface. Optionally, this interface can be associated to a VRF in the `vrf` subsystem to terminate a tunnel overlay network into a VRF domain. The underlay used by any tunnel is currently limited to the implicit default VRF, though.

The optional `local-ts` and `remote-ts` options define Traffic Selectors for the tunnel, subject to negotiation with the VPN gateway. Both options default to `0.0.0.0/0`, letting the VPN gateway narrow the selectors for IPv4 networks. A list of up to sixteen[v4.5.0] smaller subnets each can be proposed to limit the traffic to certain subnets, where `local-ts` defines source addresses and `remote-ts` defines destination addresses when sending, and destination/source addresses respectively when receiving over this tunnel.

The `vpn` subsystem periodically sends latency probes within an established tunnel to check its liveness and estimate latencies of individual uplinks used by a tunnel for load-balancing decisions. By default, latency probes are sent to the external VPN gateway address, as resolved from the `gateway` option. Some setups with a more restrictive set of `remote-ts` selectors may not cover this address, however, and require an explicit internal address of the VPN gateway to send latency probes to, defined by the `probe` option. The sending of latency probes can be completely disabled for a tunnel by setting the `probe` option to the empty string, effectively disabling any load-balancing, latency control or QoS functionality.

The `uplinks` section of each tunnel defines the underlay network uplink interfaces to consider for use by this tunnel. Tunnels are established redundantly over all uplinks, and traffic sent over this tunnel is load-balanced over usable uplinks. Each defined uplink takes the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `prio` | integer | Uplink priority group | No |
| `quality` | integer | Uplink quality base | No |

| Option | Type | Description | Mandatory |
|---|---|---|---|

The `prio` option defines a priority group, defaulting to `10`, and all uplinks with the same priority form a group. The group with the lowest `prio` value having usable uplinks is active, and only active groups are considered for load-balancing. Only if a group has no usable uplinks, the next higher group of uplinks sharing the same `prio` is considered for load-balancing. Within the same active `prio` group, traffic is load-balanced based on the quality of each uplink. The quality is estimated from different criteria, including lower layer signal quality and measured latency. The `quality` parameter defines the base quality of an uplink, defaulting to `100`. To the base quality of an uplink, the measured quality is added for the load-balancing decision, but the base `quality` can be used to weight some uplinks more than others. Example:

```
vpn:
  tunnels:
    home:
      gateway: home.example.org
      identity: router-xy@example.org
      xfrmi: xfrm0
      local-ts:
      - 10.0.1.0/24
      - fc00:1234::/64
      remote-ts:
      - 10.0.0.0/24
      - ::/0
      probe: 10.0.0.1
      uplinks:
        mbm0: {}
        lan0:
          prio: 5
          quality: 200
```

#### Bypass Rules

All IP traffic received by a Mobile Router for local processing or for forwarding is subject to IPsec processing. If the selectors of a VPN tunnel match received traffic, it is encapsulated and sent over the overlay network to the associated VPN gateway. Sometimes it is useful to exempt some traffic explicitly for IPsec processing, and this can be configured by rules in the `bypass` subsection. Bypasses can be either configured on a subnet or on a network interface basis.

For subnet based bypasses in the `subnet` `bypass` section, each named bypass rule takes a list of up to 16 CIDR subnets. If both the source and the destination address of a packet matches one of the subnets in a single rule, the packet is bypassed from any IPsec processing. Similarly, the `interface` `bypass` section takes named rules containing a list of interfaces. If traffic is coming in from and leaves over interfaces that are both listed under a single rule, it is excluded from IPsec processing. Example:

```
vpn:
  bypass:
    subnet:
      my-subnet-rule:
      - 10.0.1.0/24
      - 192.168.0.0/16
    interface:
      my-ifname-rule:
      - lan0
      - lan1
      - br1
```

#### Latency Control

Many uplinks, especially those over cellular wireless networks, suffer from significant bufferbloat. If such an uplink is put under high load, the excessive buffering along the path results in excessive latency, so far that such an uplink gets almost unusable for interactive applications. For traffic that is forwarded over tunnels, Mobile Routers can effectively reduce the latency by throttling traffic sent from either end and avoid excessive buffering along the path. By default, an RTT of $500$ ms is targeted for any uplink. Targeting lower latencies can be configured, but this may come at a cost of reduced throughput, as some buffering is required to properly saturate links, especially if they use a wireless shared medium. The `latencies` section of the `vpn` subsystem takes per-interface configuration sections, each accepting the following option:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `target` | integer | Target latency to regulate to, ms | No |

If the `target` latency is undefined, it defaults to $500$ ms. Example:

```
vpn:
  latencies:
    mbm0:
      target: 400
    lan0:
      target: 100
```

#### QoS Classes

Based on the active latency measurements run on tunnels and the active send rate adaption algorithm used to target a configured latency, the onway VPN solution can estimate the available bandwidth provided by a dynamic (wireless) uplink end-to-end. Based on this real-time bi-directional link capacity assumption, both the Mobile Router and VPN gateway can apply QoS bandwidth reservation for defined traffic classes, so that critical traffic can be prioritized over less important flows.

Under the `qos` section of the `vpn` subsystem, multiple arbitrarily named traffic classes can be defined. These classes match traffic by Layer 3 and Layer 4 selectors, and apply to all defined tunnel instances. Each class takes the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `selectors` | list[object] | Selectors to match traffic for this class | Yes |
| `absolute` | integer | Absolute bandwidth reservation, in kbit/s | No |
| `relative` | integer | Relative bandwidth reservation, in % | No |

QoS traffic classes are defined for a list of traffic patterns, where each pattern optionally defines source and destination subnets and an optional destination port. Source and destination can be on either end of the tunnel, and each packet sent from any VPN endpoint is matched bidirectionally against any source/destination direction (so it does not matter who initiated the connection). Each `selector` takes these options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `src` | string | CIDR subnet for packet source match | No |
| `dst` | string | CIDR subnet for packet destination match | No |
| `tcp` / `udp` | object | Layer 4 protocol match | No |

If given, the Layer 4 protocol match has to additionally nest a `dst-port` integer option, which defines the TCP/UDP destination port to match on.

For each QoS class, a rate can be specified to be either absolute or relative to the estimated available total rate. If both absolute and relative rates are given, the smaller of the two rates is applied to the class. For traffic matching the class, the system tries to reserve the bandwidth given the current uplink conditions, symmetrically in upstream and downstream directions. Further, traffic is limited to the given rate, so traffic from other classes (or not matching an explicitly specified class) get their calculated share of the available uplink. Example:

```
vpn:
  qos:
    my-audio:
      selectors:
      - src: 10.0.1.0/24
        dst: 172.16.5.12/32
        udp:
          dst-port: 4567
      absolute: 512
```

#### Cluster Mode

The `vpn` subsystem `clusters` section configures a cluster mode for Mobile Routers. Multiple router nodes joining such a cluster implement a virtual gateway under a shared Layer 3 gateway address. Clients using this address as their next-hop transparently send their traffic to one of the nodes in the cluster, which forwards the traffic over one of its VPN tunnels. This basically allows to extend the multipath uplink support in the VPN solution to extend to multiple physical distinct routers, without requiring explicit support from clients. This can be used to combine the bandwidth of all uplinks of each individual Mobile Router to a virtual uplink and at the same time provides hardware redundancy. The technology used by the cluster implementation uses a pool of MAC addresses to answer ARP requests for the shared IP with MAC addresses for the responsible cluster. A custom synchronization protocol is used to negotiate MAC responsibility between nodes.

The `clusters` section takes a set of arbitrarily named cluster mode configurations, each taking the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `ifname` | string | Interface name to serve clients on | Yes |
| `sync-ifname` | string | Cluster sync interface | No |
| `clusterip` | string | Shared virtual cluster IP to use | Yes |
| `secret` | string | Shared secret for cluster node authentication | Yes |
| `mac-count` | integer | Number of cluster MACs to use | No |
| `rebalance` | bool | Set for flow-granular load-balancing | No |
| `tunnels` | list[string] | Tunnels to sync in this cluster | Yes |

A node may join one or multiple virtual clusters. For all nodes joining a cluster, they must share the same values for the following options:

- The wrapping cluster configuration section name
- `clusterip`
- `secret`
- `rebalance`

The `ifname` option defines the interface to provide the virtual router service under. The `clusterip` is virtually provided on that interface and must not be installed by the `addr` subsystem or other means. Clients use the `clusterip` address as a next-hop to make use of the virtual gateway.

**Note**

The `clusterip` address is not an ordinary address and can be used for the sole purpose of a virtual router. Providing other services on that address is not supported. ICMP echo requests to this address are answered[v3.18.0] by the node responsible for the client, but only from the client serving interface specified with `ifname`. Due to technical limitations, ICMP echo responses are reflected to the MAC address the request was received from, which implies that a sender must be either in the same Layer 2 or an intermediate routing must be symmetrical.

If a `sync-ifname` is provided, the inter-cluster synchronization protocol is run on this interface, and all nodes must have the `sync-ifname` on the same Layer 2 network. If omitted, cluster synchronization uses the interface defined in the `ifname` option. The `secret` option defines a shared secret, equal on all nodes for the same cluster, authenticating synchronization protocol messages. The `mac-count` option defines the size of the cluster MAC address pool used, and ultimately defines the load sharing granularity for serving clients.

If the `rebalance` option is set, traffic is re-balanced to provide per-flow granular load-balancing for single clients; by default, the same node handles all traffic for a single client. Re-balancing additionally calculates node responsibility based on per-packet Layer 4 information. A node receiving traffic that it is not responsible for mirrors it back to the serving interface after altering the destination to be the MAC address of the responsible virtual cluster node. The `tunnels` list option links tunnels to cluster instances, so that the states of these tunnels can indicate if a node is capable of forwarding traffic from specific clients. Example:

```
vpn:
  clusters:
    myclust:
      ifname: lan0
      clusterip: 10.0.1.1
      secret: strOnG-s3¢reT
      rebalance: true
      tunnels:
      - home
```

### 8.3.33 "vrf" Subsystem

The `vrf` subsystem can create different Layer 3 routing domains. Very similar to the Layer 2 domains created with bridge interfaces from the `bridge` subsystem, a VRF device creates a Layer 3 routing domain for all network interfaces associated to it. In contrast to bridges, though, interfaces not explicitly associated to a VRF reside in the implicit *default VRF* domain. IP routing of packets is restricted to interfaces residing in the same VRF domain, and crossing VRF domains is possible via VRF interconnects, only.

The `vrf` section takes subsections for a VRF interface to create and maintain. Each VRF interface section takes the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `slaves` | list[string] | Interfaces to associate to VRF | Yes |
| `table` | integer | Routing table used by VRF | Yes |
| `interconnect` | object | VRF interconnect definitions | No |

The `table` associates a unique routing table to the VRF domain, and this table identifier may be used in the `route` subsystem to manually install routes to that VRF. Other subsystems, such as `dhcp`, implicitly use the correct routing table if the interface to operate on is associated to a VRF. To make interface names intuitive, VRF interfaces must start with `vrf`, followed by the number of the table name used by that VRF. It must be in the range of 10 and 499, excluding the reserved values 220, 253, 254 and 255.

```
vrf:
  vrf10:
    slaves:
    - br0
    - xfrm10
    table: 10
    interconnect:
      ivr1: ivr0
```

#### Interconnects

The `interconnect` parameter offers the ability to interconnect two VRF domains using a pair of `ivr` *Inter VRF Routing* interfaces. Such an interconnect works between two VRF domains like a physical direct Ethernet link would between two physical routers. Ordinary forwarding routes from the `route` subsystem and explicit firewall rules in the `firewall` subsystem allow selective traffic leaking between VRF domains.

An interconnect entry consists of a pair of `ivr` interface names, where the key is the name of the local VRF domain `ivr` interface, and the value is the `ivr` interface name in the remote VRF domain name to interconnect to. The names have to be unique in a VRF configuration, and must start with `ivr`. VRF interconnects can be created in two flavours:

- Interconnecting two explicit VRF domains, or
- Interconnect an explicit and the implicit *default* VRF domain

For the first case, the target VRF domain for the interconnect is implicitly given by using a symmetrical configuration in the peering VRF interconnect using the same `ivr` interface names, in switched order. For the second case, a second VRF matching the `ivr` interfaces is just omitted; then it is assumed that the interconnect is terminated into the implicit default VRF.

**Note**

`ivr` interconnects provide the Layer 2 connectivity between routing domains, only; additional configuration is required similar to connecting two physical routers over a direct Ethernet link, namely:

- `ivr` interfaces require IP addresses on a common subnet, acting as next-hop
- VRF domains require explicit routes to route traffic from one VRF to another
- Firewalling must explicitly allow the traffic to get forwarded, in both VRF domains

This configuration is usually done in the `addr`, `route` and `firewall` subsystems. Alternatively, *Static routes between VRFs* can be used in some cases, being significantly less complex.

When traversing VRFs, multiple routing steps are usually involved, one in each VRF. A packet entering a VRF through a slave device is routed in that VRF, potentially over an `ivr` device. After passing the interconnect, the packet is routed again in the destination VRF, likely to a slave device to reach the target. In both VRF domains, routing and firewall rules work as if two physical routers would be involved.

By default, firewalling implicitly allows forwarding for ICMP traffic or if traffic has been received over IPsec. This default does not apply to traffic that is routed *into* (but not *out of*) a VRF interconnect: Such traffic requires explicit allow rules not only for standard traffic, but also for ICMP packets or traffic coming out of a VPN, ensuring a more strict default when traversing isolated VRFs.

In the example below, `vrf10` is interconnected with `vrf11` using the link pair `ivr1` and `ivr2` in each VRF, respectively. `vrf11` additionally interconnects to the implicit *default* VRF using the interfaces `vrf3` in `vrf11` and `ivr0` in the *default* VRF:

```
vrf:
  vrf10:
    table: 10
    slaves:
    - lan0
    interconnect:
      ivr1: ivr2
  vrf11:
    table: 11
    slaves:
    - lan1
    interconnect:
```

```
    ivr2: ivr1
    ivr3: ivr0
```

#### Static routes between VRFs

VRF interconnects are powerful, but complicated to configure. For a simpler mechanism allowing traffic between VRF domains, *static routes between VRFs*[v3.17.0] can be used. This works by installing static routes in the `route` subsystem, where no `via` is specified, but a route `ifname` specifying the target VRF interface name to route certain destinations to. The `table` selects the VRF domain to install the route to, allowing traffic from that VRF to the VRF specified in `ifname`. Example:

```
vrf:
  vrf10:
    table: 10
    slaves:
    - lan0
  vrf20:
    table: 20
    slaves:
    - lan1
route:
- dst: 10.0.1.0/24
  table: 10
  ifname: vrf20
- dst: 10.0.2.0/24
  table: 20
  ifname: vrf10
```

To allow bi-directional traffic between VRF domains, usually a route is required in both domains. For VRF-crossing routes, `ICMP` traffic is implicitly allowed, but other traffic requires a `firewall` subsystem rule to allow forwarding.

### 8.3.34 "`vxlan`" Subsystem

The `vxlan` subsystem installs RFC7348 VXLAN interfaces. They are used to create Layer 2 overlay networks using UDP encapsulation, and are not restricted to Ethernet interface underlays like VLAN interfaces from the `vlan` subsystem. VXLAN interfaces use multicast addressing to discover unknown Layer 2 targets or transport broad/multicast traffic, but use unicast addresses once Layer 2 peers are known.

The `vxlan` section takes subsections for VXLAN interfaces to create and maintain. Each interface section takes the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `vxlanid` | integer | VXLAN identifier to use | Yes |
| `parent` | string | Parent interface to operate on | Yes |
| `group` | string | Multicast address used for VXLAN group | Yes |

Each peer participating in the same Layer 2 VXLAN overlay must use the same multicast `group` and `vxlanid`. To make interface names intuitive, VXLAN interfaces must start with `vxl`, followed by the `vxlanid` of that interface. Example:

```
vxlan:
  vxl10:
    vxlanid: 10
    group: 239.255.0.10
    parent: lan0
```

### 8.3.35 "wifi" Subsystem

The `wifi` subsystem configures 802.11 Wireless LAN interfaces for different purposes. It uses additional subsections for every kind of mode a WiFi interface can operate in, namely:

- `ap` subsection for Access Point operation
- `client` subsection for WiFi client configurations
- `mesh` subsection for WiFi mesh mode operation
- `mon` subsection for WiFi frame sniffing in monitor mode

Under each of these subsections, further subsections for WiFi interfaces can be nested. These WiFi interfaces can be either *physical* or *virtual*. All virtual interfaces operate on a *parent* interface, and share the same piece of hardware; Physical interfaces have names such as `wfi0` or `wfi1`. Virtual interfaces are named by the user in the form of `wfi0.1` for an interface with the parent `wfi0`. Nonetheless, the `parent` interface must be specified explicitly in the corresponding interface entry.

**Note**

Due to hardware limitations of the used WiFi modules, combining physical and virtual interfaces are subject to module specific restrictions. Please refer to the router specific manual for hardware specific notes. Most modules support a single concurrent channel only, therefore:

- When operating multiple ap and mesh interfaces on the same module, they must use the same `channel` and `cc`.
- Using multiple client interfaces or combining client interfaces and ap or mesh interfaces on the same module is not supported.

#### Access Point Mode

Interfaces to run a WiFi Access Point on are defined in the `ap` section of the `wifi` subsystem. Each per-interface subsection takes the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `ssid` | string | SSID to run on AP | Yes |
| `hidden` | bool | Set to hide the SSID in beacons sent | No |
| `channel` | integer | WiFi channel and implicit channel width | Yes |
| `power` | integer | TX power limit, in dBm | No |
| `isolate` | bool | Isolate clients on Layer 2 | No |
| `parent` | string | Parent interface for virtual interfaces | No |
| `cc` | string | Regulatory domain ISO/IEC 3166-1 country code | Yes |
| `psk` | string | WPA2 pre-shared key for protected APs | No |
| `radius` | object | RADIUS authentication configuration | No |
| `rcoi` | list[string] | List of Roaming Consortium OIs | No |

If the `hidden` option is set to `True`, the SSID is not included in Access Point Beacons and a client must explicitly scan for the configured SSID in probe requests to get a probe response from the AP.

The `channel` option specifies the $2.4$ GHz or $5$ GHz channel number to operate the Access Point on. In the $2.4$ GHz band, channels `1`, `5`, `6`, `9`, `11` and `13` define $20$ MHz wide channels. Channel `3` uses $40$ MHz by combining channel `1` and `5`, and channel `12` is $40$ MHz wide combining channel `9` and `13`. Other channels are not supported in this band. Likewise in $5$ GHz, the channel numbers also define the channel width, where the usual[14] semantics apply (`36` is $20$ MHz wide, `38` is $40$ MHz wide by combining channels `36` and `40`). $80$ MHz and $160$ MHz wide channels are currently not supported.

The `power` option defines the transmit power limit to configure, in dBm. It specifies the maximum TX power to send with; the AP may regulate the TX power down if a client is near.

If the `isolate` option is given as `True`, clients associated to the same Access Point are not allowed to communicate with each other and are isolated at Layer 2: Each client can exchange traffic with the Access Point only.

**Note**

The `isolate` flag set on Access Points affects only clients associated to the single interface. If both `wfi0` and `wfi1` serve as APs and are associated to a bridge, clients on `wfi0` still can send traffic to clients on `wfi1`. Additional isolation on the bridge using the `bridge` subsystem `isolate` flag is required to prevent this.

If the `psk` option is given, the Access Point uses *WPA2-Personal* for a protected Access Point instead of an Open System. The length of the PSK must be at least 8 characters. Alternatively, the `radius` option delegates authentication to a RADIUS server using WPA2-Enterprise with EAP, taking the following sub-options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `nasid` | string | NAS identifier | Yes |
| `client-addr` | string | IP address used for RADIUS client | No |
| `client-dev` | string | Interface name to bind RADIUS socket to | No |
| `servers` | list[object] | RADIUS servers | Yes |

A `client-addr` forces a specific address to use for RADIUS communication. A VRF for RADIUS communication can be directly or indirectly selected using the `client-dev` option, the VRF can be different from the VRF the WiFi interface is associated to. The `servers` option is a list of servers to try to authenticate clients against, each taking the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `server` | string | RADIUS server IP address | Yes |
| `secret` | string | RADIUS secret | Yes |
| `type` | string | RADIUS server type, defaults to `auth` | No |

[14] `https://en.wikipedia.org/wiki/List_of_WLAN_channels`

The `type` v4.4.0 option sets the purpose of the RADIUS server. It defines if the RADIUS server entry is used for authentication and/or accounting. When omitted, the RADIUS server will only be used for authentication. Following types of RADIUS server can be configured:

| RADIUS server type | Description |
|---|---|
| `auth` | RADIUS server is only used for authentication. |
| `acct` | RADIUS server is only used for accounting. |
| `both` | RADIUS server is used for both authentication and accounting. |

**Note**

When configuring an accounting RADIUS server, an authentication server has also to be provided for the Access Point to work.

The `rcoi` v4.4.0 option allows to configure Roaming Consortium OIs which will be advertised by the AP. An OI is a hex-encoded octet string, with a minimal length of three octets and a maximum length of 15 octets. Roaming Consortium OIs are used with OpenRoaming.

Example:

```
wifi:
  ap:
    wfi0:
      ssid: psk-test
      psk: test-p@ssw0rd
      hidden: True
      channel: 38
      power: 14
      cc: CH
    wfi0.1:
      ssid: eap-test
      channel: 38
      parent: wfi0
      radius:
        nasid: test-ap
        client-dev: vrf10
        servers:
        - server: 10.1.2.3
          secret: foobar
          type: both
      rcoi:
      - fedcba
      - 4455667788
```

#### Client Mode

Interfaces to run as a WiFi client are defined in the `client` section of the `wifi` subsystem. Each per-interface subsection takes the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `networks` | list[object] | Network candidates to connect to | Yes |
| `parent` | string | Parent interface for virtual interfaces | No |

When defining a `parent` interface, the configuration is created on a virtual interface, sharing the hardware with the parent. The `networks` option defines a list of networks with their SSID to attempt to join, and a networks entry takes the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `ssid` | string | SSID of network to join | Yes |
| `hidden` | bool | Set to use explicit scan to discover SSID | No |
| `psk` | string | Pre-shared key for WPA2-Personal | No |
| `username` | string | Username to use with WPA2-Enterprise | No |
| `password` | string | Password to use with WPA2-Enterprise | No |
| `aaa` | list[string] | Acceptable AAA TLS certificate subjectAltNames | No |
| `ca` | string | Subject Distinguished Name of AAA TLS root CA | No |
| `roaming` | integer | Aggressive roaming signal threshold, in dBm | No |

The `psk` option defines a WPA2-Personal-PSK variable length passphrase for authentication. If given, it must be at least 8 characters long. Instead of the `psk` option, both the `username` and `password` options may be defined: When both are set, *PEAP+MSCHAPv2* is expected. If the password is omitted, the `username` attribute is used to find a client certificate by subjectAltName and use it for *EAP-TLS* authentication.

When using EAP authentication with AAA backend TLS certificates (*PEAP*, *EAP-TTLS* or *EAP-TLS*), both the `ca` and the `aaa` option must be set. `ca` defines the subject field of the CA certificate under which the AAA certificate is issued; this must be the root CA of the trust chain (not an intermediate) and that CA certificate must be available on the system. The `aaa` option defines a list of subjectAltNames of the AAA server that terminates the TLS session (usually the RADIUS server).

If the `roaming` option is set, the value defines the threshold between aggressive and idle background scanning for WiFi networks. If the option is not set, no WiFi roaming is performed unless the connection fails. If the current signal quality is below (has a higher negative value) than the given value in dBm, a background scan for WiFi networks is performed every 30s. If the signal quality is better than the configured value, background scanning is reduced to once every 5 minutes, reducing the influence to the active connection. WiFi roaming is initiated whenever a better Access Point is found. Examples:

```
wifi:
  client:
    wfi0:
      networks:
      # Open System Network, no authentication
      - ssid: Public
        hidden: True
        roaming: -60
      # WPA2-Personal, with PSK authentication
      - ssid: Protected
        psk: open-the-gates
    wfi1:
      networks:
      # WPA2-Enterprise using PEAP + MSCHAPv2 username/password
      - ssid: Secure
        username: Tester
        password: t3st
```

```
      ca: C=CH, O=Example Org, CN=WiFi CA
      aaa:
      - aaa.example.org
    # WPA2-Enterprise with EAP-TLS and client certificates
    - ssid: Certs
      username: client.example.com
      ca: C=CH, O=Example Org, CN=WiFi CA
      aaa:
      - aaa1.example.org
      - aaa2.example.org
```

For WPA2-Enterprise, having both the `ca` and `aaa` properties configured is strictly required to allow proper verification of the server certificate.

#### Mesh Mode

Interfaces to run as WiFi mesh peer are defined in the `mesh` section of the `wifi` subsystem. Each per-interface subsection takes the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `ssid` | string | SSID of mesh network | Yes |
| `channel` | integer | WiFi channel and implicit channel width | Yes |
| `cc` | string | Regulatory domain ISO/IEC 3166-1 country code | Yes |
| `power` | integer | TX power limit, in dBm | No |
| `psk` | string | SAE pre-shared key for protected mesh network | No |
| `parent` | string | Parent interface for virtual interfaces | No |

The `channel` option defines the frequency band to look for and communicate with mesh peers, $20$ MHz wide channels are supported, only. The `power` option defines the transmit power limit to configure, in dBm. If a `psk` option is given, SAE (Simultaneous Authentication of Equals) is used for authenticating mesh peers; it must be at least 8 characters long. When defining a `parent` interface, the configuration is created on a virtual interface, usually requiring the use of the same channel as the parent and siblings. Example:

```
wifi:
  mesh:
    wfi0:
      ssid: TestMesh
      channel: 40
      cc: CH
      power: 16
      psk: s3cuRe-for-sUre
```

All peers must share the same `ssid`, `channel`, `cc` and `psk` settings to form a mesh network. The mesh is compliant to 802.11s and currently uses HWMP path selection.

## 8.4 Networking Configuration Conditions

### 8.4.1 "vpn" Condition

The `vpn` condition can be used in a `conditional` statement and is fulfilled if a VPN tunnel configured in the `tunnels` section of the `vpn` subsystem has been successfully established over at least one uplink. This condition is useful to activate other subsystem configurations once backend connectivity over VPN is available, such as providing a WiFi Access Point only if it is actually usable. It uses the following option:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `name` | string | Name of VPN tunnel to monitor | Yes |

The tunnel `name` refers to the section name in the `vpn` subsystem `tunnels` section. Example:

```
- if:
    vpn:
      tunnel: guest
  then:
    wifi:
      ap:
        wfi0:
          ssid: Free-WiFi
          channel: 40
```

### 8.4.2 "io-state" Condition

The `io-state` [v4.0.0] condition can be used in a `conditional` statement and is fulfilled if an I/O pin of a GPIO module is in the *high* state. This condition is useful to activate other subsystem configurations when an external signal or button is active, such as providing a WiFi Access Point on the push of a button. It uses the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `module` | string | Name of the module to monitor | Yes |
| `pin` | string | Name of the pin of the module to monitor | Yes |

The `module` and `pin` parameters refer to target hardware specific names. Example:

```
- if:
    io-state:
      module: g403-slot3
      pin: 11
  then:
    wifi:
      ap:
        wfi0:
          ssid: Free-WiFi
          channel: 40
```

### 8.4.3 "geofenced" Condition

The `geofenced` v4.1.0 condition can be used in a `conditional` statement and is fulfilled if the router position determined over GNSS is located within a geofencing zone. Geodata for geofencing must be provided by bundles[15], and geometries are referenced by the `geofence-zone` property attached to geometries. The condition uses the following options:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `zones` | list[string] | List of geofencing zones to match | Yes |

The `zones` parameter takes a list of zone names, and the condition is fulfilled if the currently measured GNSS position is in any of the specified zones. Zone associations stay active when the GNSS position gets unavailable due to a loss of a GNSS fix.

For actions to be taken before the first GNSS fix is acquired, or when a GNSS fix is lost, the following special pseudo-zones are available:

| Pseudo-zone | Description |
|---|---|
| `awaiting-fix` | While waiting for the initial GNSS fix after boot, a router is not associated to any defined zone, but instead to this special pseudo-zone. This allows to configure conditional actions that are only active while waiting for the first GNSS fix. |
| `no-fix` v4.3.0 | This pseudo-zone is active if no GNSS fix is available and is left when a new GNSS fix is received. If the GNSS receiver can still provide a position during the GNSS fix loss (e.g. from dead-reckoning), this pseudo-zone isn't active. |

Example:

```
- if:
    geofenced:
      zones:
      - CH
      - LI
      - awaiting-fix
  then:
    modem:
      slots:
        mbm0:
        - 2
  else:
    modem:
      slots:
        mbm0:
        - 1
```

[15] https://support.onway.ch/onway-router/pubdoc/latest/geo-data.html

### 8.4.4 "remote" Condition

The `remote` [v4.3.0] condition can be used in a `conditional` statement. The remote condition can be controlled over the `conditions-v1` rumpl service API. A management application on the backend can dynamically activate or deactivate the configuration section. The remote condition `name` has to be unique. Setting the condition to `true` activates the `then` section. Setting it to `false` deactivates the condition and applies the `else` section. Condition states are not preserved across router restarts and their default state is `false`. It uses the following option:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `name` | string | Name of the remote condition | Yes |

Example:

```
- if:
    remote:
      name: webcam
  then:
    link:
      lan2:
        poe:
          pse: true
  else:
    link:
      lan2:
        poe:
          pse: false
```

### 8.4.5 "fan-error" Condition

The `fan-error` [v4.5.0] condition can be used in a `conditional` statement and is fulfilled if any fan of the router fails. On a fanless router or with fans that can't be monitored, it will never be fulfilled. This condition is useful to activate other subsystem configurations, such as changing the state of an I/O module pin to signal a fan failure. It doesn't have options.

Example:

```
- if:
    fan-error: {}
  then:
    io:
      state:
        g403-slot3:
          "24":
            state: high
  else:
    io:
      state:
        g403-slot3:
          "24":
            state: low
```

### 8.4.6 "service-error" Condition

The `service-error` [v4.5.0] condition can be used in a `conditional` statement and is fulfilled if the state of any of the given service check groups fails. Once all given service check groups are up and running again, the condition isn't fulfilled anymore. The condition is not fulfilled until a first service check has completed. This condition is useful to activate other subsystem configurations, such as changing the state of an I/O module pin to signal the failure of a monitored system. It uses the following option:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `groups` | list[string] | List of service check groups to monitor | Yes |

**Note**

For the condition to work properly, the provided groups have to be configured in the "service-check" Subsystem.

Example:

```
- if:
    service-error:
      groups:
      - wifi-aps
      - fis-displays
  then:
    io:
      state:
        g403-slot3:
          "33":
            state: high
  else:
    io:
      state:
        g403-slot3:
          "33":
            state: low
```

## 8.5 Telemetry Configuration

The telemetry configuration defines telemetry sources to enable on a Mobile Router, and how often these sources are queried for reporting to the backend over the dedicated management channel. Sources may use caching to store collected telemetry data while a router is running, but currently offline. This allows the router to deliver cached telemetry data once it regains backend connectivity over one of its uplinks.

Telemetry sources are configured in a YAML template in the `telemetry` section. This section takes sub-sections for each telemetry source, each defining the reporting interval and the caching policy:

```
telemetry:
  source-name:
    interval: ms
    cache: entries
```

The *`source-name`* is a name of the source as below, whereas *`ms`* is the measurement interval in milliseconds, as integer. *`entries`* defines the number of entries to cache in-memory if no backend connection is currently available.

Some telemetry sources support *Event-based* delivery, that is, the data is not reported in a fixed interval, but when the state of a telemetry source changes. Event based reporting is enabled by omitting the `interval` option or by specifying it as `0`. If a source supports event based reporting is indicated in the table below. Some sources even support event based reporting only, setting an interval is not allowed.

| Source | Description | Events |
|---|---|---|
| `can-data` | Collected CAN bus frames | No |
| `can-fms-data` v4.1.0 | Vehicle FMS CAN bus information | No |
| `cpu-usage` | CPU usage information | No |
| `disk-info` v4.1.0 | Disk static hardware information | Yes |
| `disk-health` v4.1.0 | Disk health information | Yes |
| `disk-usage` v3.10.0 | Disk storage partition usage information | Yes |
| `fan` v4.3.0 | Fan speed measurements | Yes |
| `geofencing` v4.3.0 | Geofencing active zones | Yes |
| `gps-location` | GPS location data | No |
| `gps-satellites` | List of GPS satellites in view | No |
| `gps-velocity` | GPS velocity data | No |
| `hostname` v3.10.0 | Hostname information | Yes |
| `hw-info` v4.0.0 | Extra hardware information | Yes |
| `ip-addr` v3.18.0 | Layer 3 IP addresses of interfaces | Yes |
| `link-state` v3.10.0 | Physical link state (lower-state, speed, duplex) | Yes |
| `link-stats` | Link usage and error statistics | No |

| Source | Description | Events |
|---|---|---|
| `mac-addr` | MAC addresses of physical interfaces | Yes |
| `mem-usage` | System memory usage information | No |
| `modem-connection` | Modem network and connection state (MNC/MCC, IP connectivity etc.) | Yes |
| `modem-info` | Modem hardware, firmware and driver information | Yes |
| `modem-signal` | Modem signal levels and quality (RSSI etc.) | No |
| `modem-sim` | Modem SIM card state and information | Yes |
| `neigh-addr` v4.2.0 | Neighbour entries information | Yes |
| `network-tests` | Test results for configured network traffic tests | No |
| `os-version` | Operating System version, active/fallback system state v3.10.0, uptime information v3.10.0 | Yes |
| `service-check` v3.13.0 | Monitoring of external devices and services | Yes |
| `service-check-groups` v4.6.0 | Monitoring and local evaluation of external devices and services, organized in groups | Only |
| `temperature` v4.1.0 | Temperature measurements | Yes |
| `voltage` | Supply/Ignition voltage measurements | No |
| `vpn-stats` | Multipath VPN tunnel uplink quality and usage statistics | No |
| `vpn-state` | Multipath VPN tunnel state and load sharing | No |
| `vpn-tunnel` v4.3.0 | Multipath VPN tunnel uplink statistics | No |
| `vpn-events` v4.3.1 | Multipath VPN tunnel uplink events | Only |
| `wifi-ap-stations` | WiFi associated stations in AP mode | No |
| `wifi-cli-connection` | WiFi connection info in client mode | No |
| `wifi-mesh-peers` | WiFi connection info in mesh mode | No |
| `wifi-probes` | Collected WiFi probe request packets | No |

An example for a `telemetry` configuration is provided below. It collects CPU usage statistics in an interval of 10 seconds, and memory usage statistics in an interval of 1 minute. The caching policy for both sources total to 5 minutes for their corresponding interval. The modem connection state is reported upon changes, and change events are cached for up to ten entries when offline.

```
telemetry:
  cpu-usage:
    interval: 10000
    cache: 30
  mem-usage:
    interval: 60000
    cache: 5
  modem-connection:
```

```
  cache: 10
```

**Note**

As the onway Mixer does not provide telemetry data to the user, the data format details for telemetry reporting are currently not documented herein.

## 8.6 Bundle Configuration

A Mobile Router device configuration `bundle` section defines the bundles a Mobile Router is intended to use. A `bundle` is a read-only SquashFS filesystem image which is automatically synchronized to and mounted on the router. It provides files to use by router applications, such as static HTTP content, container filesystem or Virtual Machine base images. The `bundle` section is optional and can configure multiple bundles; each one uses a unique name and takes the following option:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `storage` | string | Storage location identifier | Yes |

The `storage` option selects the location where the bundle is stored. It references an internal disk partition, such as:

- `INTERNAL` for builtin internal flash storage with limited capacity
- `SSD:1` for the first partition of a router supporting only a single internal SSD
- `SSD2:3` for the third partition on the second internal SSD

Example:

```
bundle:
  my-bundle:
    storage: INTERNAL
  my-other-bundle:
    storage: SSD1:3
```

Referenced bundles must be available on the CEP systems the router configuration is deployed to. Creating and publishing bundles is out of scope of this documentation. Bundles can be referenced in the Mobile Router `netconfig` Networking Configuration, where the mounted bundle content can be referenced in storage locations using the `BUNDLE:` prefix followed by the bundle name.

Example:

```
  container:
    mycontainer:
      location: BUNDLE:my-bundle
      image: my-bundle-content
    ...
```

# 9 VPN Gateway Specific Configuration

In addition to the `certs` section shared with other device types, a VPN Gateway device takes an additional `vpn` configuration section as specified in the following chapter.

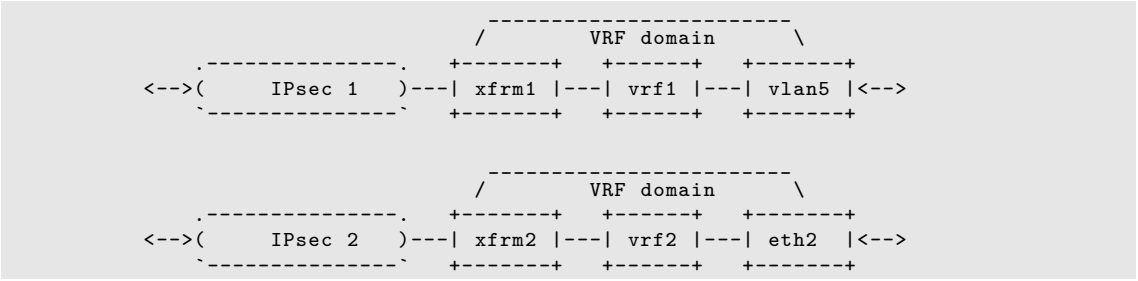
In contrast to the Mobile Router devices, Mixer does only deploy configuration to VPN Gateway devices, but no software images. Therefore, software updates for VPN Gateways need to be performed manually when required.

## 9.1 VPN Service Configuration

The onway VPN gateway is a VRF aware IPsec implementation using the IKEv2 key exchange protocol. It builds upon Linux and can terminate different tunnels into different VRF domains. It is specifically designed to work with onway Mobile Routers. The gateway assists routers in using multiple uplinks concurrently and measuring tunnel quality, and it enforcing QoS rules in the direction towards routers.

### 9.1.1 Traffic Separation

Traffic separation on the VPN gateway is implemented using XFRM interfaces bound to VRF domains. Multiple IPsec tunnels can be terminated into the same XFRM interface. Each XFRM interface is associated with exactly one VRF interface. Each VRF interface may have multiple additional interfaces to route from/to. These interfaces can be Linux VLAN interfaces or physical (or Hypervisor provided) interfaces.




The non-VRF routing domain uses one or more interfaces facing to the Internet, and usually a default route. This routing domain must handle IKE and encrypted ESP traffic, but never sees decrypted traffic. Routing and interface setup for the non-VRF routing domain is not handled by the Mixer provided configuration; the Linux distributions networking tools are used for their configuration.

Within a VRF routing domain, only plain traffic is routed. Traffic coming out of an IPsec tunnel enters the VRF domain over the associated XFRM interface. It is then routed via the routing table of the associated VRF interface. Setup of XFRM, VRF and uplink interfaces along with the required addresses and routes is provided by the configuration managed through Mixer.

### 9.1.2 Clustering

To provide redundancy at a system and network level, VPN gateways can form a cluster of nodes. BGP can be used to announce node availability and priority to both the external network in the non-VRF domain (front-door), but also to the internal networks in each VRF.

Separate BGP sessions are used to peer with routers in each VRF. Both iBGP and eBGP are supported on either network.

### 9.1.3 "vrfs" – VRF Configuration

The VPN gateway's main configuration items are VRF domains, including their associated interfaces. Under each VRF, tunnels get specified that terminate into that VRF. Each defined VRF has an implicit XFRM interface, an associated list of tunnels and a list of uplink interfaces to route traffic from/to. Optionally, a per-VRF BGP configuration enables BGP announcements to routers on the internal network.

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `vrf` | integer | Unique VRF instance table ID | Yes |
| `isolate` | bool | Isolate clients from each other | No |
| `interfaces` | object | Interfaces to create in or move into this VRF | No |
| `tunnels` | object | IPsec tunnels configuration | Yes |
| `bgp` | object | BGP announcement configuration | No |

By default, traffic coming out of a tunnel is rejected when the destination is also reachable over the tunnel. This prevents a client to reach other clients by using the gateway as a hub. When setting the `isolate` property in the per-VRF section to `False`, client isolation is disabled in this VRF, and clients may send traffic to each other over the central VPN gateway. The `vrf` table ID specifies the routing table associated to the VRF, which must be in the range of 1 and 65535, excluding the reserved values 220, 253, 254 and 255.

#### "interfaces" Objects

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `addrs` | list[string] | IP address with prefix to install, CIDR notation | Yes |
| `routes` | object | Route destination with next-hop address | Yes |
| `vlan` | object | VLAN interface configuration | No |

The `interfaces` section contains objects with the actual interface names to create in or move into that VRF domain, along with addresses and routes to set up. In the `routes` section, the key is the IP route destination subnet, and the value the next-hop gateway IP address. The next-hop must be reachable in the VRF, for example by being in one of the address prefixes given on one of the interfaces.

If the `vlan` section is missing, the interface refers to an existing physical interface, which is moved into the VRF domain.

#### "vlan" Object

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `vlanid` | integer | VLAN ID of VLAN to create on parent interface | Yes |
| `parent` | string | Name of parent interface to create VLAN on | Yes |

If the interface has a `vlan` subsection, a VLAN interface is created on the specified `parent` with the given `vlanid`. For consistent naming, the name of the interface to create must match the concatenation of the `parent` and the `vlanid`, separated by a dot.

**"`tunnels`" Objects**

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `local` | object | Tunnel local end configuration | Yes |
| `remote` | object | Tunnel remote end configuration | Yes |

Under the `tunnels` section, one or multiple IPsec configurations can be defined. `local` specifies properties for the near end of the tunnel, `remote` the properties of the far end. The `local` and `remote` subnets define the IPsec policy the tunnel shall carry traffic over. The remote subnet is usually further narrowed for each client individually by embedding the whitelisted subnets in the clients certificate `blocks` section `certs` configuration. The `remote` subnet also implicitly defines the routes added over the XFRM interface to properly route over the tunnel.

**"`local`" Object**

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `id` | string | IKE identity to use with gateway | Yes |
| `subnets` | list[string] | Local CIDR subnets for this tunnel | Yes |

Tunnel subnet configurations use certificate-based authentication. The `id` option defines the local IKE identity the gateway uses. For this identity, a certificate is required containing this identity in its `subject-altnames`, defined in the global `certs` certificate configuration section.

**"`remote`" Object**

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `ca` | string | Intermediate CA subject or subjectAltName | Yes |
| `subnets` | list[string] | Remote CIDR subnets for this tunnel | Yes |

To force a client into the correct VRF, the associated tunnel must be selected by the IKE responder. In most cases this is done by the client certificates issuing intermediate CA. That CA is specified in the `ca` property, which refers to an intermediate CA subject. When using distinct CA per VRF, this provides a simple mechanism to divide clients into VRFs. Alternatively (or additionally), distinct `id` values can be used by the same client to select a specific VRF to terminate a tunnel into, under the assumption that it has a certificate under that remote `ca`. Other mechanisms for tunnel configuration and implicit VRF selection are possible, but currently not implemented.

**"bgp" Object**

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `as` | integer | Local BGP Autonomous System number | Yes |
| `router-id` | string | Local BGP Router ID (IP address) | Yes |
| `neighbours` | list[object] | Remote BGP peers | Yes |
| `networks` v1.0 | list[string] | Arbitrary prefixes to announce | No |

The optional `bgp` section enables BGP announcements to the internal network for established VPN connections under this VRF. The `as` defines the Autonomous System, and the `routerid` the BGP router id for the local peer. The `networks` option takes a list of arbitrary prefixes to announce to all `neighbours` (using the same address family as the prefix) using the priority based on the node's state.

**"neighbours" Objects**

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `addr` | string | Remote BGP peer IP address | Yes |
| `as` | integer | Remote BGP peer Autonomous System | No |
| `auth-key` v1.3 | string | TCP-MD5 BGP authentication key | No |
| `path-prepend` v1.0 | bool | Use AS Path Prepending for prefix priorities | No |

Under `neighbours`, a list of remote peers can be specified to connect to, each receiving prefix announcements over BGP. The BGP sessions run within the VRF the `bgp` section is defined in, and announces prefixes for tunnels terminated into that VRF, only. If the `as` option is given for a neighbour and differs from the local `as`, eBGP is used instead of iBGP to communicate with that neighbour. Configured IPv4 neighbours receive IPv4 prefixes over an IPv4 BGP session, whereas IPv6 neighbours receive IPv6 prefixes over an IPv6 BGP session, only. If the `auth-key` option is set, it defines a TCP-MD5 password to authenticate the BGP session with. The neighbour must configure the same key to successfully establish the BGP TCP session.

If the `path-prepend` option is set to `True`, prefix priorities are announced by using BGP AS Path Prepending, refer to *External BGP Configuration* for details.

**Example**

Two VRF domains are created, one for `tenant-x` and one for `tenant-y`. The `vrf` property defines a unique number for that VRF domain, which is also used for VRF/XFRM interface names and the routing table associated with the VRF (`xfrm10` and `vrf10` for `tenant-x`).

```
vrfs:
  tenant-x:
    vrf: 10
    interfaces:
      ens192:
        addrs:
        - 10.0.1.1/24
        - fc00:1234::1/96
        routes:
```

```
        0.0.0.0/0: 10.0.1.10
        '::/0': fc00:1234::10/96
  tunnels:
    guest:
      local:
        id: vpn.tenant-x.example.com
        subnets:
        - 0.0.0.0/0
        - ::/0
      remote:
        ca: C=CH, O=Example AG, OU=Guest, CN=Guest CA
        subnets:
        - 172.16.0.0/22
        - fc00::/7
tenant-y:
  vrf: 12
  isolate: false
  interfaces:
    ens192.505:
      vlan:
        vlanid: 505
        parent: ens192
      addrs:
      - 10.0.2.1/24
      routes:
        0.0.0.0/0: 10.0.2.10
  tunnels:
    management:
      local:
        id: vpn.tenant-y.example.com
        subnets:
        - 0.0.0.0/0
      remote:
        ca: C=CH, O=Example AG, OU=Mgmt, CN=Mgmt CA
        subnets:
        - 172.24.0.0/14
  bgp:
    as: 64512
    router-id: 10.0.2.1
    neighbours:
    - addr: 10.0.2.10
      as: 64513
      auth-key: mysecret
      path-prepend: True
    networks:
    - 172.16.1.0/24
```

**Note**

Overlapping subnets in different VRF domains must not be used, it is currently not supported.

### 9.1.4 "bgp" – External BGP Configuration (Front-Door)

To implement system redundancy using multiple VPN gateway nodes, BGP can be used to announce an IP address shared between nodes (often called loopback address) via BGP to a

next-hop router. Strict master/backup roles are used, and the primary node announces the prefix for the shared IP address using a higher *local-pref* compared to the backup node (or a lower *MED* when using eBGP).

A node runs periodic health-checks, and if such a check fails, switches into a failed state. The *local-pref* is then reduced, so that a failing master node announces the shared address with a lower preference compared to the backup node; the backup node then starts to actively terminate tunnels.

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `interface` | string | Interface to speak BGP on | Yes |
| `addrs` | list[string] | IP addresses with prefix, CIDR notation | Yes |
| `as` | integer | Local BGP Autonomous System number | Yes |
| `router-id` | string | Local BGP Router ID (IP address) | Yes |
| `role` | string | BGP role, `master` or `backup` | Yes |
| `neighbours` | list[object] | BGP peers to announce `addrs` to | Yes |
| `check` | object | Local node health checks | No |
| `flush` v1.0 | integer | Time to defer IPsec tunnel flush, in s | No |

The `as` and `router-id` options define the local BGP peer configuration. The `addrs` section takes a list of addresses shared between nodes. These addresses get installed on the interface using the prefix length given, and are announced as loopback addresses over BGP.

**Note**

The prefix length given for addresses in `addrs` is used for installing the address, and implies the subnet size of the locally attached Layer 2 network. For shared loopback addresses, using `/32` (or `/128` for IPv6) is usually recommended. Over BGP, only a `/32` (or `/128` for IPv6) prefix is announced for these addresses, regardless of configured prefix length.

The `role` can be either `master` or `backup` and defines the base priority to announce. These are currently fixed, and when using iBGP, the announced *local-pref* priorities are:

- `master` in good state: `200`
- `master` in fail state: `100`
- `backup` in good state: `150`
- `backup` in fail state: `50`

When using eBGP, the *MED* equals to `250` minus the *local-pref* used with iBGP:

- `master` in good state: `50`
- `master` in fail state: `150`
- `backup` in good state: `100`
- `backup` in fail state: `200`

If the `path-prepend` v1.0 option is used (see below), both on iBGP and on eBGP, prefix priorities are managed using BGP AS Path Prepending. The local AS is repeated in the AS Path depending on the node state:

- `master` in good state: AS Path length `1`

- `master` in fail state: AS Path length `5`
- `backup` in good state: AS Path length `3`
- `backup` in fail state: AS Path length `7`

The same priorities get applied to the prefixes announced on the per-VRF internal BGP sessions, and are updated on fail state conditions.

When the node priority changes, the IKE daemon terminates all IPsec tunnels. This triggers a tunnel re-setup from the clients, handled by the current best node. With BGP, there may be some convergence time before new connection attempts actually land on the highest priority node. To avoid receiving these reconnects on a just degraded node, the flush of IKE tunnels can be deferred by the number of seconds given in the `flush` v1.0 option. It defaults to `0` seconds delay, and a value of `-1` disables IKE tunnel flushing.

**"`neighbour`" Object**

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `addr` | string | Remote BGP node IP address | Yes |
| `as` | integer | Remote BGP node Autonomous System | No |
| `auth-key` v1.3 | string | TCP-MD5 BGP authentication key | No |
| `path-prepend` v1.0 | bool | Use AS Path Prepending for prefix priorities | No |

The `neighbours` option takes addresses and AS identifiers for routers to peer with; if `as` is given and is different from the local `as`, eBGP is used instead of iBGP. If the `auth-key` option is set, it defines a TCP-MD5 password to authenticate the BGP session with. The neighbour must configure the same key to successfully establish the BGP TCP session.

If the `path-prepend` option is set to `True`, prefix priorities are announced by using BGP AS Path Prepending. If set so, the *MED* is omitted on eBGP, and for iBGP the *local-pref* uses a fixed value of `100`. AS Path Prepending is not recommended on iBGP, though, as BGP by default does not allow the own AS in AS Paths.

**"`check`" Objects**

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `script` | string | Shell command to check the local node state | Yes |
| `interval` | integer | Interval to execute the script, in seconds | Yes |
| `timeout` | integer | Timeout for script, in seconds | No |
| `fall` | integer | Times the script needs to fail to enter failed state | No |
| `rise` | integer | Times script needs to succeed to recover state | No |
| `defer` v1.0 | integer | Times to defer script execution after configuration | No |

The `check` section defines checker scripts to run periodically to determine the state of the local node. Each arbitrarily named `check` entry takes a `script` with a shell command to invoke, an `interval` in seconds to execute that script, and an optional `timeout` in seconds after which a check script is deemed failed. The `fall` option defines the number of times a script must fail consecutively before entering the fail state, and the `rise` option defines

the number of times a script must succeed before recovering from a fail state; both options default to `1`. A script has failed if its exit code is non-zero or if it times out.

If the `defer` option is omitted, the script is invoked immediately after the configuration has been applied. A `defer` value defers the first invocation of the check script by that number of intervals, which is useful if a check script may fail a few times after the network has been configured.

**Example**

The top-level `bgp` section defines the external BGP configuration to use. Multiple shared IP addresses are supported on the BGP speaking interface.

```
bgp:
  interface: ens160
  addrs:
  - 192.168.1.5/24
  - fc00:ab::1/64
  router-id: 192.168.1.2
  as: 64512
  neighbours:
  - addr: 192.168.1.1
    as: 64513
    auth-key: "very!secret#secret"
  check:
    ping-nexthop:
      script: ping -c1 192.168.1.1
      interval: 3
      timeout: 1
      fall: 3
      rise: 5
  role: master
```

### 9.1.5 "`firewall`" – Firewall Configuration

The `firewall` section can contain two lists of firewall rules. The `input` subsection for IPv4 rules, and the `input6` subsection for IPv6 rules. These are installed to the appropriate filter INPUT chain. No other tables or chains are currently supported. The unnamed rule objects contained in both lists use the same structure apart from the different address types, which is defined as follows:

| Option | Type | Description | Mandatory |
|---|---|---|---|
| `src` | string | Source IP address/CIDR subnet to match | No |
| `dst` | string | Destination IP address/CIDR subnet to match | No |
| `proto` | string/integer | Layer 4 protocol name or number to match | No |
| `matches` | object | Extended matching rules | No |
| `target` | object | Rule target: `ACCEPT`, `REJECT` or `DROP` | No |

By default, all incoming traffic that is not IKE/IPsec or ICMP is dropped, unless it matches a stateful firewall connection initiated from the local host. The `matches` option defines Linux Netfilter match extensions. To not interfere with the implicit rule-set, it is advised to use simple `udp` and `tcp` matches and their `srcport` and `dstport` match options, only.

> **Note**
>
> There is no implicit allow rule for incoming SSH connections. If the system is managed using SSH, an explicit rule is required to keep it accessible.

**Example**

Packets coming from source address `1.2.3.4` to UDP destination port 444 are accepted, as are any TCP packets coming from the subnet `2.3.4.0/24`. All IPv6 packets from the `fc00:abcd:/64` subnet are also accepted.

```
firewall:
  input:
  - src: 1.2.3.4
    proto: udp
    matches:
      udp:
        dstport: 444
    target:
      ACCEPT: {}
  - src: 2.3.4.0/24
    proto: tcp
    target:
      ACCEPT: {}
  input6:
  - src: fc00:abcd:/64
    target:
      ACCEPT: {}
```

# 10 Contact Information

onway ag
Stauffacherstrasse 16
CH-8004 Zürich
Switzerland
Tel: +41 55 214 18 30
info@onway.ch
www.onway.ch