nodewatcher Documentation

Release 2.0

wlan slovenija

October 20, 2014

Contents

nodewatcher is one of the projects of wlan slovenija open wireless network. Its main goal is the development of an open source network planning, deployment, monitoring and maintanance platform with emphasis on community.

CHAPTER 1

Contents

1.1 Installation

The procedure of installing your own instance of *nodewatcher* platform follows.

Note: we are assuming that you are running an UNIX-like operating system.

1.1.1 Warning Regarding Database Backend

nodewatcher assumes working support for transactional savepoints in the database backend. **This is only supported in PostgreSQL version 8.0 or higher** and therefore this is the only database that is supported by *nodewatcher*.

The system will still work with MySQL and SQLite but some features regarding error handling and validation may cause unexpected results and even data corruption! Do not use them for production deployment. You have been warned.

1.1.2 Prerequisites

The following (Debian, Ubuntu ...) packages are required:

- python (>= 2.6, >= *3.0 not supported*)
- python-django (1.4.x)
- python-rrdtool (with rrdtool >= 1.3.5)
- python-lxml
- python-yaml
- python-memcache
- fping
- graphviz
- memcached
- python-psycopg2
- python-django-picklefield
- python-django-south
- python-pyparsing

- python-anyjson
- python-pymongo
- python-geoip
- python-dnspython
- beanstalkd

The following Python modules are required:

- celery (2.5.3)
- django-celery (2.5.5)
- django-registration (newer than revision 6d2e42a13cb6, > 0.7)
- django-phonenumber-field
- python-aprmd5
- pybeanstalk

The following PostgreSQL extension is required:

- ip4r extension

Install them all with a single command:

```
sudo aptitude install python python-django python-rrdtool \
python-lxml python-yaml python-memcache fping graphviz memcached python-psycopg2 \
python-django-picklefield python-django-south python-pyparsing python-anyjson \
python-pymongo beanstalkd python-geoip python-dnspython
```

1.1.3 Getting Source

Use `default` branch which contains stable *nodewatcher* source from the project repository. Get it using this command:

```
hg clone http://dev.wlan-si.net/hg/nodewatcher
```

1.1.4 Configuration

There are few preconfigured settings files already provided for you to use as a template/base:

- `settings.py` – default settings file for local development
- `settings_wlansi.py` – settings file for local development customized for *wlan slovenija* network
- `settings_production.py` – settings file for production instance used in *wlan slovenija* network

They are all in `frontend` directory. `settings_production.py` extends `settings_wlansi.py`, which in turn extends `settings.py`. You can start testing *nodewatcher* immediatelly by using `settings.py` configuration file. But if you want to modify settings, the recommended way is to make a similar structure, extending and overriding provided settings files. In this way you will not have repository conflicts when provided settings files will get updated. And if you extend then properly, you will get good defaults automatically for new settings.

When starting django, make sure to set the `DJANGO_SETTINGS_MODULE` environment variable or `--settings` command line option so that it points to your settings file. Note that your copy of the settings file will not be updated by the source-code repository.

The `settings.py` settings file contains a lot of commented-out or disabled options (and features). This also means some dependencies are not required when using it. On the other hand `settings_production.py` uses and enables everything.

1.1.5 Runing Development/Testing Instance

Once you have all prerequisites and *nodewatcher* itself, you can run it from its `frontend` directory:

```
./manage.py runserver
```

You can then open it in your browser at http://localhost:8000/.

If you want to see how *wlan slovenija* instance looks like, run instead:

```
./manage.py runserver --settings=frontend.settings_wlansi
```

Setting Up Dummy Data

It is often useful to have some real data in your database for local development or to see how the system works with filled data. Currently we provide a cleaned-up (no passwords or other private data) dump of our database for such purposes. The dump is created daily and can be retrieved from our server. As it is made from newest repository version you should probably always update your local version also to newest version prior using this data (otherwise incompatibilities in data models can happen).

In the dump archive there are two separate components. One is a `data.json` file which is a sanitized database dump of our production setup and the other is the `graphs` directory that includes some static graphs generated by our setup.

Run in the `frontend` directory:

```
./manage.py loadtestdata
```

This will download dump archive, unpack the `graphs` directory to the `static` directory and the `data.json` to the `frontend` directory and prepare and populate database with dump data from `data.json` file.

1.1.6 Setting Up a Production/Clean Environment

You really **must use PostgreSQL** (see warnings above) so you have to configure it in Django settings file. You should also disable all debugging options. You can simply use `setting_production.py` as a template/base for your settings file. You will also need to create a file named `secrets.py` into which you put settings you do not want to have public (and by mistake pushed to the code repository). Here are some suggestions what you can put there:

- `DB_PASSWORD`
- `SECRET_KEY`
- `GOOGLE_MAPS_API_KEY`

Then for clean/empty environment you prepare database with (in the *frontend* directory):

```
./manage.py preparedb
```

It will also ask for initial administrator user data.

IP Pools

In the database you have to define your project and IP pools to be able to register nodes. You can add them for example with following SQL queries:

```
INSERT INTO nodes_pool
 (family, network, cidr, status, description, ip_subnet, default_prefix_len,
  min_prefix_len, max_prefix_len)
 VALUES(4, '10.88.0.0', 18, 0, 'Test Pool', '10.88.0.0/18', 27, 26, 28);
```

The pool in this example is `10.88.0.0/18` by default prefixes of length `/27` are allocated to nodes, but allocation of sizes `/26` through `/28` (inclusive) is also allowed.

Values are:

- `family` – should be 4 as we do not support IPv6 yet
- `network` – network address of your pool
- `cidr` – size of your pool (prefix length)
- `status` – should be 0 when first creating a toplevel pool
- `description` – nice description of the pool
- `ip_subnet` – shold be in network/cidr format
- `default_prefix_len` – default prefix length allocated to nodes
- `min_prefix_len` – min (numerically) prefix length to allow
- `max_prefx_len` – max (numerically) prefix length to allow

DNS Zones

In order to setup the DNS zones for the projects you will currently have to manually add the top-level zones into the database and then configure your DNS resolver. The instructions below apply to `bind` and you should have some experience with setting up DNS servers. First you need to create a zone by executing a command like:

```
INSERT INTO dns_zone
 (zone, owner_id, active, primary_ns, resp_person, serial, refresh, retry, expire,
  minimum)
 VALUES('xx.wlan', 1, true, 'ns1.xx.wlan.', 'dns@wlan-xx.net.', 1, 10800, 3600,
  604800, 38400);
```

Values are:

- `zone` – should be the zone's DNS name
- `owner_id` – currently unused, should be the administrators uid which is usually 1
- `active` – set to true for active zones
- `primary_ns` – DNS name of the primary nameserver
- `resp_person` – e-mail of DNS admin in hostname notation
- `serial` – current serial number, should be set to 1 when creating a zone
- `refresh`, `retry`, `expire`, `minimum` – see DNS documentation

After creating a zone you should also create some basic records in order for the zone to work properly:

```
INSERT INTO dns_record
 (zone_id, name, ttl, type, data, mx_priority)
 VALUES('xx.wlan', '@', 38400, 'SOA', 'xx.wlan.', 0);

INSERT INTO dns_record
 (zone_id, name, ttl, type, data, mx_priority)
 VALUES('xx.wlan', '@', 38400, 'NS', 'ns1.xx.wlan.', 0);
```

The top-level zone (in our example it is called `wlan`) must be configured as a zone in your resolver. An example configuration follows:

```
$TTL 38400
wlan.           IN      SOA     a.root-servers.wlan. dns.wlan-xx.net. (
                        1
                        10800
                        3600
                        604800
                        38400 )

; Root nameservers for this zone
wlan.                   IN      NS      a.root-servers.wlan.
a.root-servers          IN      A       10.x.y.z

; Subdomain delegation
xx                      IN      NS      ns1.xx.wlan.
ns1.xx.wlan.            IN      A       10.x.y.z

; Domain for test DNS checks
dns-test.wlan.          0 IN    A       127.0.0.1
```

Then you have to configure your DNS resolver to fetch some zones dynamically from the nodewatcher database. This can be done in `bind` by configuring the DLZ plugin in your `named.conf`. Sample configuration is as follows:

```
dlz "wlanXX" {
  database "postgres 1
  {host=localhost dbname=nodewatcher user=nodewatcher password=YOURDBPASSWORD}
  {SELECT zone FROM dns_zone WHERE zone = '$zone$' AND active = true}
  {SELECT ttl, type, case when type = 'TXT' then mx_priority || ' ' || '\"' || data || '\"' when type
};
```

Note: On some older `bind` versions keyword parameters to queries should be encased in `%` and not `$` (so you would use `%zone%` instead of `$zone$`).

Projects

```
INSERT INTO nodes_project
 (name, description, pool_id, channel, ssid, ssid_backbone, ssid_mobile,
  zone_id, captive_portal, geo_lat, geo_long, geo_zoom)
 VALUES('ArborMesh', 'Example project on the Moon', 1, 6,
  'open.example.net', 'open.example.net-backbone', 'open.example.net-mobile',
  NULL, true, 46.05, 14.5, 13);
```

Values are:

- `name` – name of the project, for example, city of the network
- `description` – nice description of the project
- `pool_id` – default IP pool

- `channel` – default channel used
- `ssid` – SSID used in this project
- `ssid_backbone` – SSID used for backbone nodes in this project
- `ssid_mobile` – SSID used for mobile nodes in this project
- `zone_id` – DNS zone id (NULL if DNS capabilities of ''nodewatcher" are not used)
- `captive_portal` – should the nodes in this project have captive portals?
- `geo_lat` – default location of the map when adding a new node (latitude)
- `geo_long` – default location of the map when adding a new node (longitude)
- `geo_zoom` – default location of the map when adding a new node (zoom)

And then you have to link pool with the project (of course with proper id values):

```
INSERT INTO nodes_project_pools(project_id, pool_id) VALUES(1, 1);
```

Running Web Server

For production deployment read Django documentation on the subject. **Django's development web server is not suitable for production use.**

Running Data Collection Daemon (Monitor)

Django web interface is just an interface to the database. To populate and update it with real data from the network you have to run also a monitoring daemon.

Run the monitor using command (in `monitor` directory):

```
./monitor.py --path=.. --settings=frontend.settings_production
```

You also need to install `olsrd-mod-txtinfo` plugin on some node in the network and configure it via OLSR configuration file (also note the node's firewall configuration). By default monitor expects OLSR `txtinfo` plugin on localhost. This and other options you can configure in Django settings file.

Checking OLSR `txtinfo` Plugin

You can check that the `txtinfo` plugin is working by issuing:

```
telnet 10.x.y.z 2006
Trying 10.x.y.z...
Connected to 10.x.y.z.
Escape character is '^]'.
```

Then type `GET` and press enter. This should output something like:

```
HTTP/1.0 200 OK
Content-type: text/plain

Table: Links
...lots of data...
Connection closed by foreign host.
```

This means that the plugin is working properly.

Simulation of Monitor Data

To simulate monitor data you should set `MONITOR_ENABLE_SIMULATION` to `True` in your setting file. In this case the whole network is simulated and no node with OLSR providing the data feed is required. **This may not be suitable for all test scenarios.** Simulation data should be provided in `simulator/data` directory.

The latest simulation data that can be retrieved from this location. Simply unpack it into `simulator/data` directory.

Optional Data Archival System

nodewatcher supports an optional data archival system so all graphed data is also stored in a non-RRD database. We currently use MongoDB for this store due to its schemaless document nature and fast operations. In order to use this feature, you need to install and configure a MongoDB instance and then configure *nodewatcher* via `DATA_ARCHIVE_*` directives in `settings.py`. You will also need the `pymongo` Python driver for MongoDB.

You should familiarize yourself with MongoDB operations, durability limitations and proper deployment modes. Documentation is accessible via the above link.

On-demand Graph Feneration

All graphs are generated on-demand when requested by the web frontend to reduce I/O load on monitor runs. Because this requires additional configuration/setup, default configuration has the on-demand graph generation disabled (and therefore no graphs are displayed). **You should configure this after you already have a working monitor setup.**

On-demand graph generation requires a working installation of a message broker (for details see Celery documentation). We use MongoDB for this purpuse in production via the `mongodb` backend. If you already have a working MongoDB installation (it is also used for the optional data archive system) you simply need to set `BROKER_HOST` and `BROKER_PORT` to proper values for your MongoDB setup. Check `settings_production.py` file for an example.

After you have the broker set-up you also need to run `celeryd` task dispatcher in the background. You can do this simply via `manage.py` as follows:

```
./manage.py celeryd -l info -c 4 --maxtasksperchild=50
```

For production systems you will probably create an init script for starting up the dispatcher. Be sure that the user under which the deamon is executed has privileges to write to `GRAPH_DIR`. The last thing to do is to set `ENABLE_GRAPH_DISPLAY` to `True` in your settings file.

Firmware Image Generator

After you have configured all of the above components you might also want to enable the firmware image generator daemon. As the whole process is based on OpenWrt, you first need to build the imagebuilders for our firmware. The procedure below assumes creation of a new directory, but symlinking or building the imagebuilders on another system (as this is a very CPU and IO intensive process) is also possible.

Setup the needed directories and compile the imagebuilders using the following commands (if you are doing it remotely, you really should run this inside a `screen` session so that compiling is not interrupted if your session is disconnected):

```
mkdir build
cd build
hg clone http://dev.wlan-si.net/hg/old/openwrt-nw openwrt-200901
hg clone http://dev.wlan-si.net/hg/nodewatcher
```

```
cd nodewatcher/generator
./build_all_generators.sh
```

This will take a long time and will heavily load your CPU and IO. It is only needed to rebuild the imagebuilders when updating to a new version of the firmware. After the above process is completed without errors you must create the user with an username as configured with `IMAGE_GENERATOR_USER` in your setttings file. It should be in the same-named group. This will be the user the process will run under. You also need to setup a local instance of the `beanstalkd` daemon that should run on `127.0.0.1`, port `11300` (refer to beanstalkd documentation for details). After that, you may run the image generator using the following commands:

```
./gennyd.py --path=.. --settings=frontend.settings_production --destination=/srv/www/packages.foonet
```

The `destination` argument should reflect your `IMAGES_BINDIST_URL` configuration in your settings file. This means that it should point to the physical directory that is backed by the URL. The directory must be writable by the `IMAGE_GENERATOR_USER` user. After you have configured everything and the generator is running, you should set `IMAGE_GENERATOR_ENABLED` to `True`.

Theming

It is possible to configure distributed *nodewatcher* theme or even develop your own custom theme, see *Theming*.

1.2 Theming

It is possible to configure distributed *nodewatcher* theme or even develop your own custom theme. The theme is composed of four layers:

- favicon, logo and possible other graphical elements of the network using *nodewatcher* installation
- icons and other graphical elements to visualize *nodewatcher* elements (like nodes)
- SCSS files to generate CSS used for layout styling
- Django templates for XHTML layout

1.2.1 Favicon and Logo

Favicon and logo can be changed through settings. There are `NETWORK_FAVICON_URL` and `NETWORK_LOGO_URL` settings to set customized favicon and logo URLs.

1.2.2 Overriding CSS

For small changes you might just want to override CSS in the browser, loading your changes after the provided CSS files. So no changes to distributed files are needed. You will just need to provide a `head.html` Django template file which should then include your own CSS file. This file should be somewhere in Django template search path, but before the distributed (empty) one.

You should use this approach really just for minor modifications. If you happen to find yourself overriding a lot of CSS all around then this will probably be unmaintainable in the future when distributed *nodewatcher* theme will change and you will have to manually keep up with all the changes. In this it is best to extend distributed *nodewatcher* theme through SCSS.

1.2.3 Changing CSS

Color scheme and other styling modifications can be achieved by changing the CSS of the theme. A new CSS style can be created using basic knowledge of CSS, however it is recommended that you first try to extend and modify an existing basic theme. **Do not modify the actual CSS** in this case as it is generated automatically using the Compass authoring framework. The source SCSS files are located in the `static/scss` directory. One can check the Compass documentation for more information about Compass.

Assuming that you have now Compass installed on your system, here are the instructions how to compile the default *nodewatcher* theme. Compass files are organized into projects. To compile our project go to `static/scss` and type:

```
compass compile
```

This will update CSS files in `statics/css`.

To use development settings (adds line comments with source information) use:

```
compass compile -e development --force
```

During the actual development running this command may become annoying so the handy way to compile CSS on the fly is to run the following command in a separate terminal:

```
compass watch -e development --force
```

In this case Compass will pool for changes to the source files and compile them when needed. You can now focus solely on creating the theme.

It is recommended that you use our SCSS files as a base and extend them through capabilities provided by Compass. So create your own Compass configuration file and directory with SCSS files alongside the `default` directory and add your changes there, using our files as a base.

SCSS Files

The syntax of the SCSS files is very similar to the one of the CSS, however, there are some handy features that make development easier. Some basic examples can be found here (note that there is also an older syntax called SASS that should not be confused with SCSS). More examples on mixins provided by Compass can be found on a Compass website.

1.2.4 Changing Django Templates

More drastic changes in layout can be made by modifying the Django templates. In this case some knowledge of Django templating system is required. You can then override specific template files from `frontend/templates` directory by copying them in some other directory, modifying them and then adding the directory path to the templates directory list `TEMPLATE_DIRS` in your `settings.py` file (before the default templates directory). Templates are modular so it should be easy to change only parts you need.

Again, be careful not to change templates in a way to be hard to maintain them with future *nodewatcher* versions. If you need more modular templates for your needs, feel free to open a ticket with request.

1.3 Telemetry

The telemetry provider component enables any node to report its operational attributes via a simple HTTP-based hierarchical key-value format. Provider is modular, easily extendable, and is split into multiple OpenWRT packages as follows:

- `nodewatcher-core` provides the basic framework for developing telemetry modules, it is required by all other packages.
- `nodewatcher-clients` provides client-related monitoring (in combination with `nodogsplash` and `olsr-mod-actions` plugin).
- `nodewatcher-watchdog` is responsible for performing periodic testing of node's network sanity and for attempting to recover from weird situations.
- `nodewatcher-solar` provides power regulator telemetry via the solar package.
- `nodewatcher-digitemp` provides 1-wire thermometer telemetry via the digitemp package.

1.3.1 Installation

Installing these packages should be just a matter of adding our repository to `/etc/opkg.conf`:

```
src/gz wlansi http://bindist.wlan-si.net/profiles/PLATFORM
```

where PLATFORM is one of the supported platforms (currently `atheros`, `brcm24` and `ar71xx`). Then installing packages should be just a matter of an `opkg install`.

1.3.2 Source Code

The source code for OpenWRT packages providing telemetry are available on GitHub. Feel free to make pull requests with additional providers you might need, and especially to request improvements to the framework provided in core package for easier and modular development of such providers.

1.3.3 Telemetry Provider for Servers

Servers often do not run OpenWRT, so a Python-based CGI script is available which provides similar telemetry in a compatible format.

1.3.4 Structure

Remote invocation scripts are installed into `/www/cgi-bin` and should be accessible via HTTP URL in the form of `http://x.y.z.w/cgi-bin/nodewatcher` (where `x.y.z.w` is the node's primary IP address).

Individual telemetry modules are installed into `/etc/nodewatcher.d/`, there is also an example module available in our repository (note that this module is not installed).

1.3.5 Format

The format used by our provider is a simple text format. Lines starting with a semicolon (`;`) are comments and should be ignored by parsers. Any non-comment line is composed of two parts separated by the first left-wise colon (`:`). Left part is denoted as *key* and the right part as *value*.

All keys form a hierarchy. Namespace atoms are ASCII strings that match the regular expressions based on their position in the hierarchy:

- Top-level namespace atoms must match `[A-Za-z-]+`. In addition, if a top-level namespace atom contains a capital letter from the range A-Z, the whole atom must be capitalized. Capitalized top-level atoms are reserved for internal use.

- Namespace atoms on lower levels must match `[a-z0-9-]+`.

Each key may contain multiple namespace atoms separated by dots (`.`), for example a valid key is `wireless.radios.ath0.bssid`. The format of value is currently not defined and should be considered of a per-key type defined by the module that outputs it. Value must not contain a newline, as newlines separate key-value pairs.

Currently the only special namespace is `META`. It contains information about installed modules and their versions. This information is available in `META.modules.*.serial` and is of integer type. Dots in module names are replaced with dashes (`-`) when they are used as namespace atoms.

In the future this hierarchical namespace will be centrally allocated to individual *nodewatcher* telemetry providers, but currently the allocations are ad-hoc and may change.

1.3.6 Example

The following is an example of *nodewatcher* output from one of the nodes:

```
;
; nodewatcher monitoring system
;
META.version: 2
META.modules.core-clients.serial: 1
iptables.redirection_problem: 0
net.losses: 0
META.modules.core-general.serial: 1
general.uuid: 7061c9ab-2bcc-442e-b0bc-d9959b519e75
general.version: hg
general.local_time: 43987
general.uptime: 43987.78 41507.07
general.loadavg: 0.25 0.27 0.30 1/39 523
general.memfree: 14056
general.buffers: 1496
general.cached: 5044
META.modules.core-traffic.serial: 1
iface.eth0.down: 0
iface.eth0.up: 0
iface.eth1.down: 0
iface.eth1.up: 0
iface.wlan0.down: 73108492
iface.wlan0.up: 23656806
iface.edge0.down: 0
iface.edge0.up: 1909300
META.modules.core-vpn.serial: 1
net.vpn.upload_limit:
net.vpn.mac:
META.modules.core-wireless.serial: 1
wireless.radios.wlan0.bssid: 02:CA:FF:EE:BA:BE
wireless.radios.wlan0.essid: open.wlan-si.net
wireless.radios.wlan0.frequency: 2.447
wireless.radios.wlan0.mac: 54:E6:FC:F3:7F:54
wireless.radios.wlan0.rts: off
wireless.radios.wlan0.frag: off
wireless.radios.wlan0.bitrate:
wireless.radios.wlan0.signal: 0
wireless.radios.wlan0.noise: 0
wireless.errors: 0
wifi.bssid: 02:CA:FF:EE:BA:BE
```

```
wifi.essid: open.wlan-si.net
wifi.frequency: 2.447
wifi.mac: 54:E6:FC:F3:7F:54
wifi.rts: off
wifi.frag: off
wifi.bitrate:
wifi.signal: 0
wifi.noise: 0
wifi.errors: 0
```

1.4 Schema

Warning: This is an **incomplete** database schema for *nodewatcher*. Current schema summary only describes some monitoring-related data – image generator profiles (configuration) is not documented at this moment.

1.4.1 Node

Each node is represented as an instance of model Node. A node may be in any of multiple states (depending on factors specified by the monitoring system):

- **Invalid** – Node is visible in the routing tables, but is not known to *nodewatcher* (entries with this status are temporary and are removed as soon as a node is not visible anymore). Any other status means that the node is known to *nodewatcher*.
- **Up** – Node is visible in the routing tables and has replied to at least one ICMP ECHO probe (there were no duplicate packets received).
- **Visible** – Node is visibile in the routing tables but has not replied to any ICMP ECHO probes.
- **Duped** – Node is visible in the routing tables and has replied to at least one ICMP ECHO probe twice with the same sequence number.
- **Down** – Node is not visible in the routing tables.
- **Pending** – Node has not yet been seen in the routing tables since its registration with the *nodewatcher*.

In the monitoring subsystem, a node is identified by its `ROUTER-ID` within the mesh. Currently this is the main IPv4 address for the node. Each node also carries a UUID that is used as primary node identifier as it is persistent across node IP changes.

Schema is as follows:

Field	Modifiers	Description
uuid	primary key	unique node identifier (Version 4 UUID as per RFC 4122)
ip	unique observed	contains the `ROUTER-ID` for this node
name	unique	human readable node name, must be a valid DNS hostname
owner		node maintainer
location		human readable node location
project		project that this node belongs to (projects also define IP pools and DNS zones)

Continued on next page

Table 1.1 – continued from previous page

Field	Modifiers	Description
notes		maintainer-specific notes
url		URL containing further information about the node (usually pictures of node's view)
geo_lat		latitude (in decimal degrees)
geo_long		longitude (in decimal degrees)
ant_external		a boolean flag indicating that an external antenna is being used by the node
ant_polarization		enumeration of following values: • *Unknown* – polarization is not known • *Horizontal* – horizontal polarization • *Vertical* – vertical polarization • *Circular* – circular polarization
ant_type		enumeration of following values: • *Unknown* – antenna type is not known • *Omni* – an omni-directional antenna • *Sector* – a sector antenna • *Directional* – a directional antenna
system_node		a boolean flag indicating that this node provides core mesh services (like root DNS, NTP, etc.)
border_router		a boolean flag indicating that this node is a border router, that is it may redistribute external routes
node_type		enumeration of following values: *Server* – this type is usually used for non wifi nodes *Mesh* – a standard wifi mesh router *Test* – testing node that is even more experimental than others *Unknown* – nodes with `Invalid` state
redundancy_link	observed	a boolean flag indicating whether this node has a redundant VPN link to the mesh
redundancy_req		a boolean flag indicating whether the lack of a redundant VPN link should trigger a warning message
conflicting_subnets	observed	a boolean flag indicating whether any of the node's subnets are currently in conflict
warnings	observed	a boolean flag indicating whether warning messages are present for this node

Continued on next page

Table 1.1 – continued from previous page

Field	Modifiers	Description
status	observed	previously mentioned node status
peers	observed	number of active node peers according to mesh topology
peer_list	observed	many to many relation of peer links
last_seen	observed	timestamp when node was last seen
first_seen	observed	timestamp when node was first seen
wifi_mac	observed	wifi interface MAC address
vpn_mac	observed	VPN interface MAC address (observed)
vpn_mac_conf		VPN interface MAC address (configured)
channel	observed	currently used wifi channel
rtt_min	observed	RTT measurement
rtt_avg	observed	RTT measurement
rtt_max	observed	RTT measurement
pkt_loss	observed	packet loss
firmware_version	observed	firmware version used on the node
bssid	observed	node's BSSID
essid	observed	node's ESSID
local_time	observed	local node's time
clients	observed	number of clients connected via nodogsplash
clients_so_far	observed	cumulative number of clients so far
uptime	observed	current node's system uptime (not network connectivity) in seconds

Modifier description:

- *primary key* – this field is a primary identifier for the given node
- *unique* – this field must be unique among all nodes
- *observed* – this field is updated by the monitoring system

1.4.2 Link

Links represent topological connections between nodes as reported by the routing daemon. Each link has the following schema:

Field	Modifiers	Description
src	observed	source node
dst	observed	destination node
lq	observed	link quality
ilq	observed	inverse link quality
etx	observed	routing metric (these fields are currently as reported by OLSR routing daemon and are probably routing-protocol specific)

All links are of temporary nature and are kept in sync with routing topology updates.

1.4.3 Subnet

Each announced and/or registered IP subnet is represented in the database schema by the Subnet model. A subnet may be in any of multiple states (depending on factors specified by the monitoring system):

- **AnnouncedOk** – subnet is known to *nodewatcher* and is allocated to the node that is currently announcing it.
- **NotAnnounced** – subnet is known to *nodewatcher*, allocated to a specific node but the node that the subnet is allocated to is not announcing it at the moment.
- **NotAllocated** – subnet is not known to *nodewatcher*, but a node is announcing it anyway.
- **Subset** – subnet is a more specific announce of a node's subnet that has status `AnnouncedOk` (subnet entries with this status are only temporary - they are present as long as they are seen in the routing tables and are treated the same as `AnnouncedOk`).
- **Hijacked** – subnet is known to *nodewatcher* but is being announced by some node other than that to which the subnet has been allocated.

Subnet schema is as follows:

Field	Modifiers	Description
node	observed	node the subnet belongs to
subnet	observed	textual representation of the subnet
cidr	observed	subnet prefix length
description		description which may be specified by the user
allocated		a boolean flag specifying whether this has been allocated via *nodewatcher* databse
allocated_at		allocation timestamp
status	observed	previously mentioned subnet state
last_seen	observed	timestamp when subnet was last seen
ip_subnet	ip field	bitwise representation of the subnet for r-tree optimized subnet queries (used for conflict detection)

Modifier description:

- *ip field* – this field is a special r-tree indexed field implemented by the IP4R PostgreSQL extension

CHAPTER 2

Source Code, Issue Tracker and Mailing List

For development *wlan slovenija* open wireless network development Trac is used, so you can see existing open tickets or open a new one there. Source code is available on GitHub. If you have any questions or if you want to discuss the project, use nodewatcher mailing list.

CHAPTER 3

Indices and Tables

- *genindex*
- *search*