WifiService Module Description

Created on: December 23, 2024

Authors: Henk Stevens, Olaf Mastenbroek, Onno Janssen

Copyright: © 2024, Oradio Stichting

License: GNU General Public License (GPL)

Organization: Oradio Stichting

Version: 3

Email: oradioinfo@stichtingoradio.nl

Status: Development

Summary: Class for wifi connectivity services

Documentation:

https://pypi.org/project/nmcli/

https://superfastpython.com/multiprocessing-in-python/

# 1. Introduction

This module provides robust and modular management of Wi-Fi connectivity for the Oradio system. It manages the system’s ability to connect to infrastructure networks, switch to a fallback access point mode, and communicate state changes through a multiprocessing queue. The module interfaces with NetworkManager using the `nmcli` Python library and supports both infrastructure and AP modes with automated failover and recovery logic.

# 2. Class Structure and Responsibilities

The core `wifi\_service` class encapsulates the following responsibilities:  
- Maintaining internal state (`error`, `saved\_ssid`).  
- Sending structured messages (`send\_message`).  
- Connecting to Wi-Fi networks (`wifi\_connect`, `wifi\_connect\_thread`).  
- Disconnecting from networks (`wifi\_disconnect`).  
- Managing access point mode (`access\_point\_start`, `access\_point\_stop`).  
- Querying available and stored networks (`get\_wifi\_networks`, `get\_wifi\_nm\_connections`).  
- Determining current state (`get\_state`).

# 3. Detailed Method Descriptions

## \_\_init\_\_(queue)

Initializes the instance with a multiprocessing queue for message passing. Sets:

- `self.msg\_q`: reference to the queue used for sending state messages.

- `self.error`: initialized to `None`, holds the last operation error code.

- `self.saved\_ssid`: initialized to `None`, used to remember the previous network when entering AP mode.

## send\_message()

Builds a message dict containing:

- `type`: constant `MESSAGE\_WIFI\_TYPE`.

- `state`: current state from `get\_state()`.

- `error`: last error code or `MESSAGE\_NO\_ERROR`.

Logs the message at DEBUG level and places it onto `self.msg\_q`.

## wifi\_connect(ssid, password)

Initiates connection to the specified SSID:

1. Checks if already connected; if so, sends current state and returns True.

2. Disconnects any active network via `nmcli.connection.down()`; on failure, logs error and sends message.

3. Removes stale NM entry if SSID exists and is not `saved\_ssid`.

4. Adds new connection profile (AP mode if SSID equals `ACCESS\_POINT\_SSID`, else WPA-PSK config).

5. Spawns `wifi\_connect\_thread` to activate the connection asynchronously.

Returns True on successful initiation, False on immediate errors.

## wifi\_connect\_thread(new\_ssid, old\_ssid)

Runs in a background thread to:

- Activate the connection with `nmcli.connection.up(new\_ssid)`.

- On failure, attempts to restore `old\_ssid` and deletes the failed new SSID profile.

- On success, logs activation and removes `old\_ssid` if different from `saved\_ssid`.

Finally, calls `send\_message()` to report the final state.

## wifi\_disconnect()

Disconnects the current active network:

1. Calls `nmcli.connection.down(active)`; on error, sets `self.error` and sends message.

2. Deletes the profile with `nmcli.connection.delete(active)`; on failure, logs and sends message.

3. Sends a final state message. Returns True if disconnected cleanly.

## access\_point\_start(force\_ap=False)

Enters AP mode with captive portal support:

1. Checks current state; if already AP, returns True.

2. Redirects DNS via `dnsmasq` config using `run\_shell\_script()`.

3. Optionally saves `active` SSID when `force\_ap` is True.

4. Calls `wifi\_connect(ACCESS\_POINT\_SSID, active)` to configure AP.

Returns True on success.

## access\_point\_stop()

Exits AP mode and restores previous network:

1. Removes DNS redirection file.

2. If AP was active and `saved\_ssid` exists, reconnects via `wifi\_connect`.

3. Otherwise, calls `wifi\_disconnect()`.

4. Clears `saved\_ssid` and sends completion message.

## get\_wifi\_networks()

Scans and returns SSIDs of on-air Wi-Fi networks using `nmcli.device.wifi()`, excluding the Oradio AP SSID and duplicates.

## get\_wifi\_connection()

Retrieves the name of the currently active Wi-Fi connection by parsing `nmcli.connection()` and checking `GENERAL.STATE` for `activated`. Returns SSID or None.

## get\_wifi\_nm\_connections()

Returns a list of all stored Wi-Fi connection profiles by filtering entries from `nmcli.connection()` where `conn\_type` is `wifi`.

## get\_state()

Determines module state: idle, infrastructure, local-only, or AP mode:

- If no active SSID: `STATE\_WIFI\_IDLE`.

- If active equals `ACCESS\_POINT\_SSID`: `STATE\_WIFI\_ACCESS\_POINT`.

- Otherwise, tests internet reachability via `check\_internet\_connection()`:

- True: `STATE\_WIFI\_INFRASTRUCTURE`

- False: `STATE\_WIFI\_LOCAL\_NETWORK`.

## 4. State Determination and Event Messaging

The `get\_state()` method plays a pivotal role in determining the current Wi-Fi status of the Oradio system. It distinguishes between being idle (not connected), connected to a local network, or functioning as an access point. The classification depends on the results of both the currently active connection (via `get\_wifi\_connection`) and an internet check using `check\_internet\_connection()`. This modular check ensures that the Wi-Fi state remains dynamically responsive to real-time conditions.  
  
Additionally, the `send\_message()` method is used throughout the class to report Wi-Fi state changes back to a shared message queue. This decouples the Wi-Fi layer from the rest of the control architecture and allows responsive adaptation to connectivity events.

## 5. Access Point Management

The `access\_point\_start()` and `access\_point\_stop()` methods manage the Oradio access point mode, used for configuration via captive portal. When started, the DNS is redirected to a local address (`ACCESS\_POINT\_HOST`), ensuring that connecting clients are rerouted to the local web interface.  
  
The method uses shell commands to place DNS redirection settings in the appropriate NetworkManager configuration directory. Upon stopping the access point, these settings are removed, and the original Wi-Fi network (if any was saved) is reconnected.

## 6. Standalone Operation and Testing Interface

This module also includes a robust standalone testing mode, available when the script is executed directly. A command-line interface allows developers and testers to manually:  
  
- List all Wi-Fi networks on air  
- View or remove connections from NetworkManager  
- Start and stop the Oradio access point  
- Simulate Wi-Fi connect/disconnect scenarios  
  
This interactive CLI is designed for field diagnostics and offers insights into Wi-Fi functionality without needing to run the entire Oradio system.

## 7. Reliability and Error Handling

To ensure robustness, each NetworkManager command is wrapped in a try-except block. Meaningful error messages are logged via `oradio\_log`, and system state is accurately communicated back to the message queue using predefined error codes such as `MESSAGE\_WIFI\_FAIL\_CONNECT`, `MESSAGE\_WIFI\_FAIL\_AP\_START`, and others.  
  
These error indicators allow the Oradio controller or frontend to respond appropriately—for instance, prompting the user to retry or switch networks, or defaulting to access point mode.

## 8. Integration Advice

When using this module within the full Oradio system, instantiate it with a shared `Queue` for messaging and call the appropriate methods (`wifi\_connect`, `access\_point\_start`, etc.) based on user interactions or startup state. Make sure the required dependencies (`nmcli`, `network-manager`, etc.) are installed and that the system supports NetworkManager management through `nmcli`.  
  
Logging can be monitored in the Oradio logs (`oradio.log`), and system feedback will appear in the messaging queue for higher-level orchestration or UI display.

Oradio WiFi Service Module – Standalone Test Description

# 6. Standalone Test and Usage

The `wifi\_service` module includes a self-contained test suite that allows developers to verify the behavior of WiFi management functions without launching the full Oradio control system. This standalone mode is triggered when the module is run directly as a script.  
  
Upon execution, the script sets up a multiprocessing message queue and launches a background process (`check\_messages`) that listens for state and error messages generated by the `wifi\_service` instance.  
  
The user is then presented with an interactive command menu that includes the following options:  
  
0 - quit  
 Exit the test script.  
  
1 - get wifi state  
 Prints the current WiFi state. The output may be:  
 - `STATE\_WIFI\_IDLE` – no active WiFi connection.  
 - `STATE\_WIFI\_INFRASTRUCTURE` – connected to an external network with internet.  
 - `STATE\_WIFI\_LOCAL\_NETWORK` – connected to a network without internet.  
 - `STATE\_WIFI\_ACCESS\_POINT` – Oradio access point is active.  
  
2 - list on air wifi networks  
 Scans for and displays SSIDs of WiFi networks currently available.  
  
3 - list wifi networks in NetworkManager  
 Shows all stored WiFi connection profiles managed by NetworkManager.  
  
4 - remove network from NetworkManager  
 Prompts for the SSID of a stored network and deletes it from NetworkManager.  
  
5 - get active wifi connection  
 Prints the SSID of the currently active connection, if any.  
  
6 - connect to wifi network  
 Prompts for SSID and password. Initiates a connection and prints result through the message queue.  
  
7 - disconnect from wifi network  
 Disconnects from the active network and removes its profile from NetworkManager.  
  
8 - start access point  
 Configures and activates the Oradio access point. Used for captive portal mode.  
  
9 - stop access point  
 Deactivates the access point and restores the previous network connection (if saved).  
  
All transitions and error messages are passed to the message queue, providing a real-time view of the system’s behavior.  
  
Note:  
- Ensure NetworkManager is running on the system.  
- Administrative permissions may be needed for some DNS and connection changes.  
- This testing framework is especially useful for Raspberry Pi systems without desktop environments.