EVE Airplane Mode

Use Case

The Airplane mode (later renamed to "**Radio Silence**") was requested by an EVE customer. Their use case is very interesting. A truck will use an edge device with EVE OS and LTE ONLY connectivity. The device will start running their applications and at some point they will have to start using explosives. They would like to shut down all radios temporarily. During this time, the applications should keep running. It is not acceptable to restart the device during this time as there are critical applications that are running on them.

Requirements

Firstly, for safety reasons it is essential that the radio-silence functionality is managed locally. Regardless of the controller connectivity status, the operator working on the location should be able to enable/disable radios as needed.

We can assume that a keyboard, mouse and a monitor will be attached to the device. The operator should be able to change and view the radio status using these peripherals. It is not required to add any additional hardware like a physical toggle button or an LED. The radio status could be therefore managed via an application deployed on the device.

Background

Airplane mode in Linux

A typical Linux distribution uses the rfkill subsystem (RF stands for radio frequency). It is a generic interface for disabling any radio transmitter in the system. When a transmitter is blocked, it shall not radiate any power.

The rfkill core provides an API for kernel drivers to register their radio transmitter with the kernel. Most importantly, an rfkill driver implements a method `set_block` for turning the radio device on and off. User-space programs, such as the NetworkManager or the rfkill tool, communicate with the rfkill core via `/dev/rfkill` device. For example, if you enable airplane mode in a Debian Linux distribution, behind the scenes the NetworkManager tries to change the kernel rfkill state (same as running `nmcli radio wwan off`).

However, this is all just plumbing and for us much more interesting is how rfkill drivers actually enable/disable radio transmitters. Following calls to rfkill_register we can find implementations for all commonly used platforms. All of them, however, end up calling a closed-source device firmware through ACPI. Nothing we can take inspiration from.

It seems that the majority of rfkill drivers are developed for use in laptops. Firmware of industrial computers often lacks rfkill support for cellular modem (WiFi adapter typically can be disabled using rfkill). This is the case in particular for external modems connected over USB.

Airplane Mode in Android

Android does not utilize the rfkill subsystem. Instead, it glues telephony services with the radio hardware using Radio Interface Layer (RIL). It consists of a stack of two components: a RIL Daemon and a Vendor RIL. Android provides a reference Vendor RIL, based on the Hayes AT command set, that can be used as a quick start for telephony testing and a guide for commercial vendor RILs. For example, the reference RF power-switch handler (implementing `RIL_REQUEST_RADIO_POWER` request) uses `AT+CFUN=0` command to switch the modem into a minimum functionality mode (without RF). Commercial vendor RILs are likely also based on device-specific AT commands to manage their cellular modems.

Airplane mode in cellular modems

Looking at mobile broadband devices specifically (and omitting WiFi adapters for now), we can learn about airplane mode by studying the source code of the ModemManager. Interestingly, this daemon is also used by NetworkManager to control cellular modems, just not for the airplane functionality. This is probably because rfkill covers all types of devices with radio transmitters and perhaps the rfkill drivers also operate closer to the hardware than ModemManager, which uses AT commands, QMI and MBIM protocols. The ModemManager is extensible via plugins to support new devices. For most modern modems that support either QMI or MBIM protocols, there are already generic plugins MMBroadbandModemQmi and MMBroadbandModemMbim.

For QMI devices, the airplane mode is actually a *low-power* mode:

```
/**
 * QmiDmsOperatingMode:
 * @QMI_DMS_OPERATING_MODE_ONLINE: Device can acquire a system and make calls.
 * @QMI_DMS_OPERATING_MODE_LOW_POWER: Device has temporarily disabled RF.
 * ...
 * Operating mode of the device.
 */
```

Here we can see how RF can be disabled using QMI. This can be done using CLI with:

```
uqmi -d /dev/cdc-wdm0 --set-device-operating-mode low_power
```

To obtain the current operating mode, we have to use more feature-rich `qmicli`:

```
qmicli -d /dev/cdc-wdm0 --dms-get-operating-mode
```

For MBIM-enabled devices, this is how ModemManager disables radio.

The same can be done using `mbimcli`:

```
mbimcli -p -d /dev/cdc-wdm0 --set-radio-state on|off

mbimcli -p -d /dev/cdc-wdm0 --query-radio-state
```

Not all modems implement QMI or MBIM. For those there are modem-specific plugins, typically based on AT commands. As it happens, most modems can be programmed to turn off the radio using the `AT+CFUN=4` command. "AT+CFUN" means to change the phone's functionality and with the parameter "4" all receive and transmit RF circuits are disabled.

EVE `wwan` Service

Currently, EVE OS is able to manage a single QMI or MBIM enabled LTE modem. There is a separate `wwan` service, running a shell script which periodically checks the wwan connectivity and tries to (re)start and (re)connect the modem using `uqmi` and `mbimcli` tools. `wwan` service communicates with `zedbox` using files written under `/run` directory. Specifically, `nim` publishes the user configured APN into `/run/accesspoint/wwan0` and the `wwan` service publishes state data and metrics as several json files under `/run/wwan`. These are periodically read and included into device info/metrics by zedagent.

Currently, the wwan service is limited to one LTE network with one APN.

The `wwan` service behaviour can be described using the following pseudo-code:

```
wwan main:
    use hard-coded config (device, interface name, etc.)
    enable mbim, qmi kernel modules
    detect protocol (QMI or MBIM)
    repeat indefinitely:
        if connectivity check failed:
            reset modem
            wait for device registration on the network
            start network (take APN from /run/accesspoint/wwan0)
            wait for data connectivity and IP address
            configure interface and resolv.conf
        publish state data (signal strength, IP configuration) under /run/wwan
        sleep 5mins (watchdog not used for wwan)
```

EVE Radio-Silence mode Proposal

Overview

The main purpose of this document is to present a design proposal for the radio silence mode feature for EVE OS. Note that we use "Radio Silence" instead of "Airplane mode". This is because in most likely use-cases for this feature the edge node will not be actually inside an airplane and the purpose of disabled radio transmission will not be related to or limited to air travel safety. In industrial applications, the act of disabling all radio transmission for safety or security reasons is more commonly referred to as "Radio Silence".

Firstly, given that the feature should be managed locally, we propose to leverage a recently added Local Profile server. The idea was to essentially allow a locally deployed application to act as a local controller and override a (very small) subset of the device configuration received from the controller. Currently, only the "profile" field can be overridden, hence the name. However, the feature was designed knowing that it could be extended in the future with more device configuration options that need local override. The API of the Local Profile server defines endpoint `/api/v1/local_profile` from which EVE should periodically obtain a profile name and use it instead of the default profile selected by the controller. The endpoint content is a protobuf-serialized message, which makes it easily extensible and allows to add more fields for the local override. However, for radio management we will define a separate P OST endpoint, making it easier to remove/disable or to split from local profile server into a separate application if needed. For API details see "EVE API Additions" below

The behavior (and to some degree the implementation) of this feature would be inherited from the Local Profile Server.

This means that:

- The application to act as a Local Profile Server is selected via the controller
- Local profile server has to authenticate itself to EVE OS using a token
- Configuration from Local Profile server overrides the configuration from the controller (even when the Local server is not currently running /accessible/responsive). In this case, the radio silence is not actually configurable by the controller. The behavior is to act as if the feature was disabled (i.e. radios are enabled) by the controller. This means that only Local profile server can enable it and by default (i.e without local profile server app being deployed) the feature is disabled and radios are turned ON.

- The configuration from the Local Profile Server is preserved between system reboots using the `/persist` partition

The requested radio silence state would be periodically obtained by `zedagent` from the Local Profile server. The handler of this configuration option would be the `wwan` service. The file-based communication between the `zedagent`/`nim` and `wwan` service will be used to exchange the intended/actual radio state. This is described in more details below under “EVE API Additions”.

The Local Profile Server from the user perspective would be in this case an application with a UI containing a toggle button to switch the radio silence ON and OFF. Given the safety requirements of the use-case, it is reasonable to also publish the current radio state periodically up to the application for the user to check. To simplify, there will be a *single* POST API endpoint for both the radio configuration (intended state) and the radio state data (the actual state).

EVE will *periodically* POST the actual state of the radio and *may* obtain the intended state in the response from the application.

A change requested by the user through this UI would not have an immediate effect, however. If we used the same time interval as for the local profile endpoint (`timer.config.interval`), it would by default take up to one minute for EVE to even notice the change in the required radio state. More delay will be also added by asynchronous `zedbox`-`wwan` communication and the actual radio on/off operation will also take some time to take effect.

In order to decrease the latency, we propose to use a separate and shorter time period of 5 seconds for the radio POST API. And in order to simplify the interaction between the application and EVE and to avoid race conditions, EVE will not POST radio state while the configuration change is still in-progress (even if it exceeds the 5sec time interval). Radio state information published while a state change is ongoing wouldn’t have much sense/validity anyway.

The actual radio on/off operation will be done by `wwan` service using `QMI` / `MBIM` protocol as opposed to modem-specific AT commands. As it has been already mentioned, there are CLI tools available for both protocols that allow to change and read the radio power state.

Out of Scope

Implementation of the application providing the user with the radio silence ON/OFF button (i.e. the Local profile server) is out of scope of this document as well. Here we only describe the interface between the application and EVE OS and describe the implementation changes needed to be done on the EVE side.

Even though this document discusses the challenges associated with a device reboot and what effect it may have on the radio state, for this particular customer it is actually unacceptable for a device to reboot during radio silence (due to the nature of their use-case). For now we therefore do not have to have a solution of preserving the radio state immediately after booting, instead we should focus on avoiding a device reboot in the first place.

EVE API Additions

Local Profile Server (new endpoint)

`POST /api/v1/radio`

Where the radio request will carry a binary-encoded protobuf message containing the actual state of every radio:

```
message RadioStatus {
        // true if enabled AND successfully applied
        bool radio_silence = 1;
        // If the last radio configuration change failed, error message is reported here.
        // Please note that there is also a per-modem configuration error reported under CellularStatus.
        string config_error = 2;
        // for every LTE network
        repeated CellularStatus cellular_status = 3;
        // later we can add status for every WiFi network
}

message CellularStatus {
        // Logical label assigned to the physical cellular modem.
        string logicallabel = 1;
        org.lfedge.eve.common.ZCellularModuleInfo module = 2;
        repeated org.lfedge.eve.common.ZSimcardInfo sim_cards = 3;
        repeated org.lfedge.eve.common.ZCellularProvider providers = 4;
        string config_error = 10;
        string probe_error = 11;
}
```

Where `ZCellularModuleInfo`, `ZSimcardInfo` and `ZCellularProvider` are defined in `api/proto/info/info.proto` as follows:

```
enum ZSimcardState {
  Z_SIMCARD_STATE_INVALID     = 0;
  Z_SIMCARD_STATE_ASSIGNED    = 1;
  Z_SIMCARD_STATE_PROVISIONED = 2;
  Z_SIMCARD_STATE_ACTIVE      = 3;
  Z_SIMCARD_STATE_SUSPENDED   = 4;
  Z_SIMCARD_STATE_CANCELLED   = 5;
}

message ZSimcardInfo {
  // Name is a SIM card identifier. For example ICCID if available.
  // Guaranteed to be unique only in the scope of the edge node.
  string name = 1;
  // Reference to ZCellularModuleInfo.name
  string cell_module_name = 2;
  string imsi = 3;
  string iccid = 4;
  ZSimcardState state = 5;
}

message ZCellularModuleInfo {
  // Name is a module identifier. For example IMEI if available.
  // Guaranteed to be unique only in the scope of the edge node.
  string name = 1;
  string imei = 2;
  string firmware_version = 3;
  string model = 4;
  ZCellularOperatingState operating_state = 5;
  ZCellularControlProtocol control_protocol = 6;
}

enum ZCellularOperatingState {
  OPERATING_STATE_UNSPECIFIED = 0;
  OPERATING_STATE_OFFLINE = 1;
  OPERATING_STATE_RADIO_OFF = 2; // AKA radio silence
  OPERATING_STATE_ONLINE = 3;
  OPERATING_STATE_ONLINE_AND_CONNECTED = 4;
  OPERATING_STATE_UNRECOGNIZED = 5;
}

enum ZCellularControlProtocol {
  CONTROL_PROTOCOL_UNSPECIFIED = 0;
  CONTROL_PROTOCOL_QMI = 1;
  CONTROL_PROTOCOL_MBIM = 2;
}

message ZCellularProvider {
  // Public land mobile network code.
  string plmn = 1;
  string description = 2;
  // True if this is the provider currently being used.
  bool current_serving = 3;
  bool roaming = 4;
}
```

The Local profile server application may either reply with status 204 (No content), acknowledging the received state data update but not requiring any configuration changes, or return 200 (OK) and append response body with the intended radio state encoded using this proto message:

```
message RadioConfig {
        string server_token = 1;
        bool radio_silence = 2;
}
```

If the actual state and the returned intended state differ, EVE will trigger the operation of applying the intended state (e.g. disabling the radio). The outcome of this operation (i.e. the new state, with potential error message if the change failed) will be published by the next POST call. If the actual and the intended state are the same, EVE will not perform any radio state changes at least until the next POST call.

EVE API - device configuration (i.e. API between EVE and the controller)

Since we are reusing the Local profile server here, the existing configuration fields are sufficient to enable and select the application that will manage the radio state.

Specifically, these two parameters will be reused for the radio silence purposes:

```
message EdgeDevConfig {
  ...

  // local_profile_server, if set, indicates a hostname/IPv4/IPv6 address and
  // optional port number at which EVE will request for a local profile.
  // If such a local profile is retrieved, it will override the global_profile.
  // The syntax follows the usual URL server name syntax thus the following
  // are example valid strings:
  //    [fe80::1]:1234
  //    10.1.1.1:1234
  //    hostname:1234
  //    [fe80::1]
  //    10.1.1.1
  //    hostname
  // If the port number is not specified, it will default to 8888
  string local_profile_server = 28;
  // Together with a local_profile_server one can specify a
  // profile_server_token. EVE must verify that the response from the
  // local_profile_server contains this token.
  string profile_server_token = 29;
}
```

As a result, no changes/additions are needed to be implemented on the controller side. Only the UI / documentation could mention that there is this additional usage of the local profile server. Also, the current state of radio will be presented to the controller (see below) and it could be therefore displayed alongside other network state information.

EVE API - device info (i.e. API between EVE and the controller)

```
message DevicePort {
        ...
        WirelessStatus wirelessStatus = XY;
}

message WirelessStatus {
        // either LTE or WiFi (or not wireless)
        WirelessType type = 1;
        // for LTE:
        repeated CellularStatus cellular_status = 5;
        // later we may add status for WiFi
}

message ZCellularStatus {
  // Name reference (ZCellularModuleInfo.name) to the corresponding cellular module
  // from the list ZInfoDevice.cellularModules
  string cellular_module = 1;
  // Each item is a name reference (ZSimcardInfo.name) to a SIM card from the list ZInfoDevice.simCards
  // Ordered by slot numbers.
  repeated string sim_cards = 2;
  // List of available cellular service providers.
  repeated org.lfedge.eve.common.ZCellularProvider providers = 3;
  // If EVE failed to configure the cellular connection, the error is published here.
  string config_error = 10;
  // if the connectivity probing is failing, error is reported here
  // (see CellularConnectivityProbe).
  string probe_error = 11;
}
```

EVE API - device metrics (requested by the same customer)

```
message deviceMetric {
        ...
        repeated CellularMetric cellular_metric = XY;
}

message CellularMetric {
        // logicallabel of the physical device
        string logicallabel = 1;
        CellularSignalStrength signal_strength = 2;
        CellularPacketStats    packet_stats = 3;
}

// Value of 0xFFFF means that the particular metric is not available.
message CellularSignalStrength {
        int32 rssi = 1;
        int32 rsrq = 2;
        int32 rsrp = 3;
        int32 snr = 4;
}

// Collected by the modem itself and can be obtained using
// e.g. "qmicli -d /dev/cdc-wdm0 --wds-get-packet-statistics"
message CellularPacketStats {
        // NetworkStats are already defined in EVE API
        NetworkStats rx = 1;
        NetworkStats tx = 2;
}
```

Interface between `wwan` service and `zedbox`

A new `/run/wwan/config.json` will be added to submit LTE configuration from `nim` to `wwan` service, including the required radio state, e.g.:

```
{
        "radio-silence": true,
        "networks": [
                {
                        "logical-label": "lte-modem1", # logical label assigned to the physical modem device
                        "physical-addrs": {
                                # nim will specify one/some of these. With multiple LTE modems the USB address
is the most unambiguous and reliable.
                                "interface": "wwan0",
                                "usb": "1:2.3", # <bus>:[port]
                                "pci": "0000:11:00.0",
                        },
                        "apns": ["internet"],
                        "probe": {
                                "disable": false
                                "address": "1.1.1.1"  # default is 8.8.8.8
            }
                }
        ]
}
```

Note: `interface/usb/pci` will be obtained by `nim` from `AssignableAdapters` based on the `PhyLabel` of `NetworkPortConfig`.

Next a new `/run/wwan/status.json` with cellular state data will be published by `wwan` service for `nim` to read. Status file will contain SHA256 checksum of the last applied revision of `config.json`. With this, EVE will know when there are no more pending configuration changes, so that it is ready to publish radio state up to the application.

```
{
        // sha256 checksum of config.json applied at the time of publishing this info
        "config-checksum": "d14a028c2a3a2bc9476102bb288234c415a2b01f828ea62ac5b3e42f",
        "networks": [
                {
                        "logical-label": "lte-modem1",    # can be empty for modems not configured from the
controller
                        "physical-addrs: {
                                # all addresses will be filled by wwan service
                                “interface”: “wwan0”,
                                “usb”: “1:2.3”, # <bus>:[port]
                                "pci": "0000:11:00.0",
                        },
                        "cellular-module": {
                                "imei": “310170845466094”,
                                "model": "QUECTEL Mobile Broadband Module",
                                "revision": "EC21ECGAR06A04M1G",
                                "control-protocol": "qmi" | "mbim",
                                "operating-mode": "offline" | "online" | "online-and-connected" | "radio-off" |
"unrecognized"
                        },
                        "sim-cards": [
                                {
                                        "iccid": "8991101200003204514",
                                        "imsi": "313460000000001"
                                }
                        ],
                        "config-error": "",
                        "probe-error": "Failed to ping 8.8.8.8 (2 packets transmitted, 0 received, 100% packet
loss, time 1029ms)",
                        "providers": [
                                {
                                        "plmn": "310-410",
                                        "description": "AT&T",
                                        "current-serving": true,
                                        "roaming": false
                                },
                                {
                                        "plmn": "310-120",
                                        "description": "O2",
                                        "current-serving": false,
                                        "roaming": false
                                }
                        ]
                }
        ]
}
```

Lastly, wwan service will also periodically publish `/run/wwan/metrics.json` with cellular metrics:

```
{
        "networks": [
                {
                        "logical-label": "lte-modem1",    # can be empty for modems not configured from the
controller
                        "physical-addrs: {
                                # all addresses will be filled by wwan service
                                “interface”: “wwan0”,
                                “usb”: “1:2.3”, # <bus>:[port]
                                "pci": "0000:11:00.0"
                        },
                        "packet-stats": {
                                "rx-bytes": 456456,
                                "rx-packets": 1234,
                                "rx-drops": 0
                                "tx-bytes": 23485,
                                "tx-packets": 758,
                                "tx-drops": 12
                        },
                        "signal-info": {
                                "rssi": -42,
                                "rsrq": -11,
                                "rsrp": -98,
                                "snr": 56
                        }
                }
        ]
}
```

Application Behavior

Application acting as a Local Profile Server for the radio management is expected to behave as follows:

- Application provides UI on the user-side and a HTTP server with `/api/v1/radio` on the EVE-side
- The UI should allow to:
  - Toggle the radio silence intended state (ON/OFF) and as a result change the content that the HTTP server will respond with to `/api/v1/radio` POST calls.
  - Show the last received actual state and the last error if there was any (periodically POSTed to `/api/v1/radio` by EVE)
  - Show that an operation of changing the radio state is still ongoing (e.g. a “loading gif”). This starts from a moment of user changing the intended state in the UI, continues through the next POST API call (from which EVE learns the new intended state, which in this case would differ from the actual state), up to the second next POST API call with the actual radio state after making the attempt to apply the new intended config (please remember that for simplicity sake there will be no POST API calls while a change is ongoing).

`zedbox` Implementation Changes

Inside the `zedbox` process the handling of the radio silence mode will be split between `zedagent` and `nim` microservices. Communication between the Local profile server and EVE is already being done by `zedagent`. Periodically, it obtains the latest profile configuration, stores it into a file for persistence and publishes it inside `ZedagentStatus`. For the radio-silence feature we will mostly reuse the same code and add `RadioSilence` (the intended state) into the structure:

```
type ZedAgentStatus struct {
        ...
        RadioSilence RadioSilence
}

RadioSilence struct {
        Enabled           bool // from application
        ChangeInProgress  bool // from NIM
        ChangeRequestedAt time.Time // recorded by zedagent after POST call
}
```

nim will subscribe for zedagent status updates. If RadioSilence.Enabled has changed, nim will trigger the operations of switching all radios ON /OFF. This actually means to publish the new configuration into wwan (and later also wlan/rfkill) service and (asynchronously) wait for updated state data (acknowledging the latest config using a checksum). Once wwan service responds, nim will publish the new state back into zedagent using DeviceNetworkStatus, which will be also extended with the same RadioSilence structure (.ChangeRequestedAt copied from ZedAgentStatus; .Enabled and .ChangeInProgress updated by nim). Inside, the per-port NetworkPortStatus will contain WirelessStatus. zedagent will make another POST call to /api/v1/radio only after it has received DeviceNetworkStatus with RadioSilence of the same timestamp and with ChangeInProgress being false.

At the same time nim will take into account the radio state during network connectivity testing. If the radio silence is ON (or a change is in progress), DPC (device port configuration) verification will be skipped. The state of enabled radio silence will be indicated by both diag and ledmanager microservices. We could also introduce a distinct blinking count for the case of lost controller connectivity due to disabled radio(s) - e.g. 5 blinks.

Please note that there is also a timer inside EVE to reboot the device if it hasn't had controller connectivity for an entire week. Since it is not expected that radio silence will be turned ON for such an extended period of time in practice, this behavior will not change.

wwan Service Implementation Changes

The shell script of the wwan service will undergo few implementation changes. Firstly, it will stop using hard-coded configuration values and instead will wait for and read /run/wwan/config.json (written by nim). It will then keep monitoring config.json file and will apply any changes, including the radio-silence ON/OFF switch.

A secondary goal of these implementation changes is to prepare (either fully or at least partially) the wwan service for scenarios with multiple LTE modems and multiple APNs.

The new wwan service behavior can be described using the following pseudo-code:

```
wwan main:
    repeat indefinitely:
        for every LTE network:
            if /run/wwan/config.json changed/appeared:
                detect protocol (QMI or MBIM)
                reset modem (i.e. disconnect)
                (re)load config
                set operating mode (online or low-power)
            if not in the radio-silence mode (operating mode is "online") and connectivity check failed:
                reset modem if connected
                wait for device registration on the network
                start network(s) (take APN(s) from config.json)
                wait for data connectivity and IP address
                configure interface and resolv.conf
            publish state data and metrics under /run/wwan/ (power state, signal strength, modem info, etc.)
        sleep 5mins, break from sleep if /run/wwan/config.json changes/(dis)appears
```

Testing

To test the above it makes sense to implement a very basic local profile server app instance in the form of a container, which each time the POST method is handled will look for a file at a specific location with the intended radio state to apply (as ON/OFF strings or 0/1).

At the same time the application could publish the state information obtained from EVE into another file with a hard-coded path. With this behavior it will be easy to prepare test-scripts under eden.

More challenging for the automated testing is the LTE modem side. Unless we can somehow emulate a cellular modem, we will need a physical device with a modem and preferably also with a registered SIM card available in a lab.

Issues and Risks

The main issue of this proposal is that the operation of switching the radio-silence ON/OFF is handled asynchronously with a considerable delay. Whereas longer delay may be acceptable when edge device is managed from the cloud, there may be different user expectations for local management. Also, there is a risk that the user may impatiently start to toggle the radio-silence ON/OFF button, triggering configuration changes before the previous one(s) had been fully handled and potentially hitting some race condition scenarios.

In order to limit the latency and the risk of race conditions, we proposed a separate and a shorter time period between EVE calls to radio POST API. Additionally, EVE will not make radio POST calls while a change is ongoing, thus enforcing at most one configuration change to be in progress at a time. Conversely, when an radio state change is finalized, EVE may call radio POST API with radio state update immediately, instead of waiting for the timer to fire. Internally, we will also try to minimize delay in communication between microservices (zedagent, nim, wwan).

One more issue, or rather an open question, is whether we can actually combine the Local Profile server with the radio-silence mode or if there is a customer requirement to have them separated and managed from different applications. This remains to be communicated with the customer. However, by having `/api/v1/radio` and `/api/v1/local_profile` as separate endpoints, it should be possible and relatively easy to allow them to be handled by different applications.