RaveModemFactory

Software architecture

Aleksander Morgado
<aleksander@aleksander.es>

ChangeLog

Date	Version	Changes
2016-07-01	6	Minor update: • Updated section 5 rmfcli to specify that `rmfcli` is just a test tool.
2016-06-29	5	Update for RaveModemFactory 1.26: • Updated sections 1.2 Components and 3.1 Startup with information about how the TCP/IP based support works. • Added section 4.3.1, Library initialization API to specify how to select between local or TCP/IP based operations.
2015-04-02	4	Update for RaveModemFactory 1.24: • Updated section 4.3.9 Network connection API with the new `GetDataPort()` method. • Updated section 5 rmfcli with the new `--get-data-port` action.
2015-04-01	3	Update for RaveModemFactory 1.22: • Updated section 4.3.6 Power management API with the new `PowerCycle()` method. • Updated section 5 rmfcli with the new `--power-cycle` action.
2015-03-19	2	Update for RaveModemFactory 1.20: • Added section 3.9 Statistics.
2015-02-20	1	Initial version.

Table of Contents

1. Introduction

The RaveModemFactory is a **connection manager** that allows to use 3GPP mobile broadband modems (e.g. GSM, HSPA, or LTE) implementing the **Qualcomm MSM Interface** (QMI) protocol. This protocol will be available in modems with Qualcomm Gobi 3G/4G chipsets, like e.g. the Sierra Wireless MC7710 or the newer MC7304. 3GPP2 technologies (e.g. CDMA or EV-DO) are not currently supported.

The RaveModemFactory system provides a **simple** and **stateless** interface to the basic operations that can be performed with a mobile broadban modem.

One of the major advantages of the system is that the user does not need to know which kind of modem is connected to the system, or which interfaces are exposed, or how to control those interfaces. The RaveModemFactory will work with every QMI-capable modem, regardless of manufacturer or version.

The user application will just need to use the RaveModemFactory interfaces to fully interact with the modem, as well as to get a fully established Internet connection.

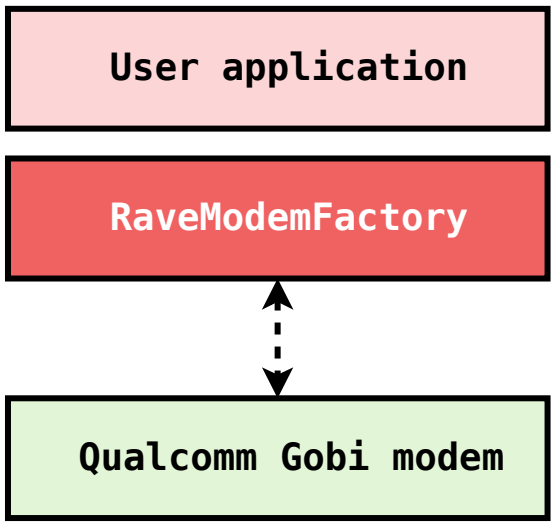

Illustration 1: High level architecture

1.1. Dependencies

The RaveModemFactory software is based on several system level tools, including:

- **Linux kernel** with the `cdc-wdm` (since 2.6.26) and `qmi_wwan`[1] (since 3.4) drivers.
- The **Glib/GIO**[2] (>= 2.32) libraries. These libraries are the core foundation libraries of the GNOME project, and provide a general-purpose set of tools for C-based applications.
- The **GUdev**[3] (>= 147) library. This library provides a Glib/GIO based interface to use the `udev`[4] device manager (e.g. for automatic device discovery).
- The **libqmi**[5] (>= 1.6.2) library. This library exposes a simple Glib/GIO based asynchronous

1 https://sigquit.wordpress.com/2014/06/11/qmiwwan-or-gobinet/
2 https://developer.gnome.org/gio/stable/
3 http://www.freedesktop.org/software/systemd/gudev/
4 http://en.wikipedia.org/wiki/Udev
5 https://sigquit.wordpress.com/2012/08/20/an-introduction-to-libqmi/

interface to use the QMI protocol with the modem. Along with the library, a `qmicli` tool will also be available, which allows to use the QMI protocol directly from command line.

- The **iproute2**[6] tools to manage network interfaces.
- The ISC **dhclient**[7] DHCP client daemon.

1.2. Components

The RaveModemFactory system is itself built as 3 different components:

- The **`rmfd`** system daemon, developed in C using the Glib/GIO library, which takes care of the actual control of the modem (including network interface and routing rules setup).
- The **`librmf`** library, developed in C++, which provides a simple interface for the actions available through the `rmfd` daemon. The communication between the `rmfd` daemon and the `librmf` library is done either through a standard Unix socket[8] or through a TCP/IP socket.
- The **`rmfcli`** tool, developed in C++, which provides a command line interface for all the operations exposed by the `librmf` shared library.

Although the command line interface is a fully functional tool, applications wanting to use the RaveModemFactory are expected to link to the shared `librmf` library.

6 http://en.wikipedia.org/wiki/Iproute2
7 https://www.isc.org/downloads/dhcp/
8 The unix socket is by default exposed at `/tmp/rmfd-server`.

2. Architecture

The following diagram depicts a typical setup of the RaveModemFactory components and how they interact with the other system tools. The user application in the diagram will use the shared `librmf` library interfaces to control the modem and gather status information from it.

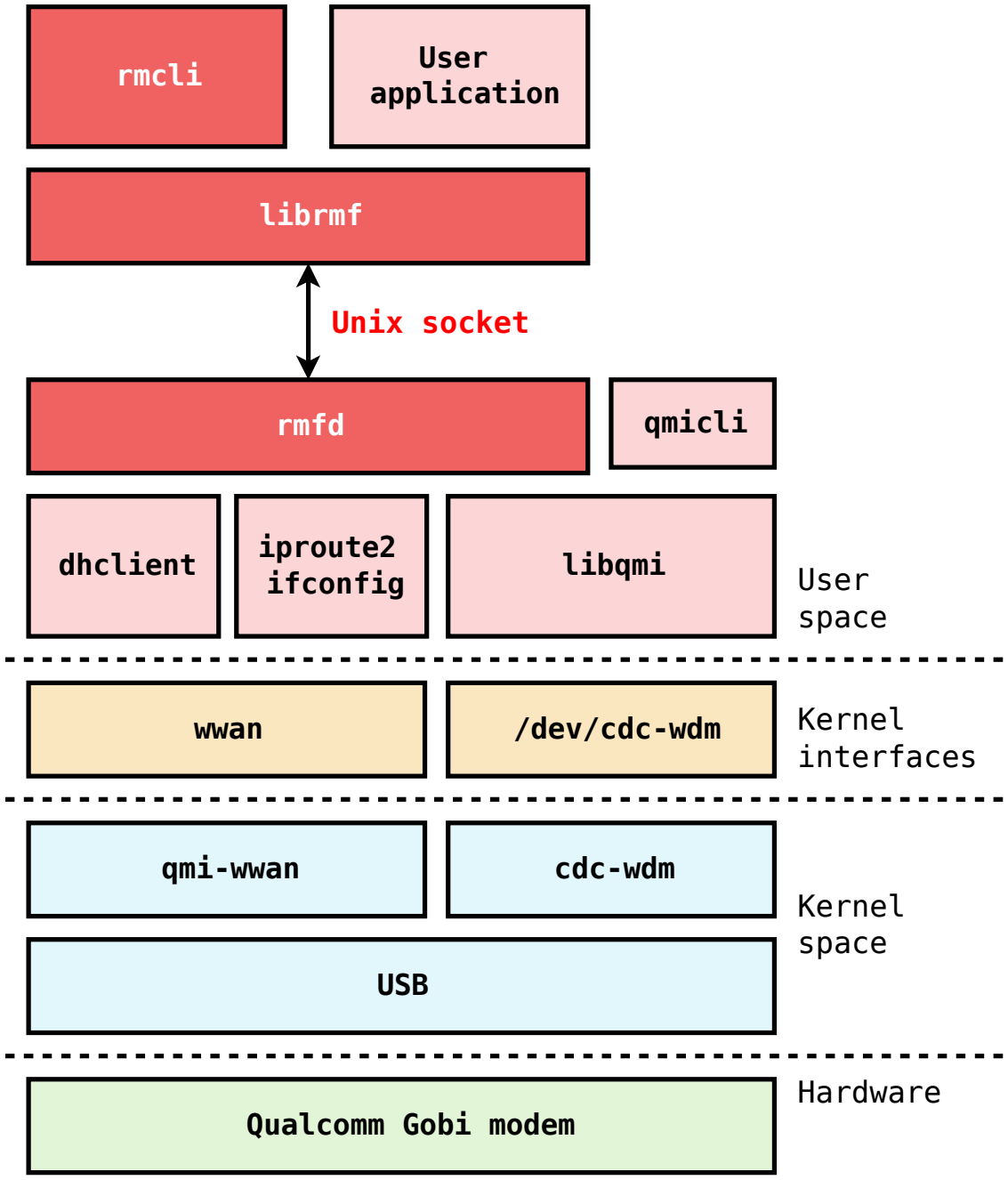

Illustration 2: Low-level architecture

The blocks in the diagram are organized by color as follows:

- Dark red: the RaveModemFactory components.
- Light red: other userspace programs and libraries.
- Orange: the modem interfaces exposed by the kernel.
- Blue: the kernel and its drivers.
- Green: the modem hardware.

3. Basic operation

3.1. Startup

The `rmfd` daemon is setup as a **system daemon** that should be automatically launched as any other service in the system (e.g. via init scripts or systemd service units). In the same way, the daemon should be shutdown when the system is going to be halted.

If no additional option is given when launching the `rmfd` daemon, it will listen by default in the local **`UNIX` socket**. For the **TCP/IP** based operation the daemon expects these additional options:

```
Usage: rmfd <option>

Application Options:
  -y, --address=[IP]     IP address where to enable the TCP listener
  -Y, --port=[PORT]      Port where to enable the TCP listener
```

3.2. Device discovery

The **kernel** is in charge of detecting the Qualcomm Gobi modem via its **USB 2.0** subsystem. Once the USB device is detected and configured, the `qmi-wwan` and `cdc-wdm` kernel drivers will take care of the QMI interfaces. For every QMI-capable USB interface, the `qmi-wwan` driver will expose a **`wwan`** network interface, and the `cdc-wdm` driver will expose an associated **`/dev/cdc-wdm`** character device.

Once the network and character devices are exposed by the kernel, they will also get notified via the **`udev`** device manager. The `rmfd` daemon listens to udev events, and will therefore automatically discover that a QMI modem is available.

3.3. Device initialization

Before using the mobile broadband modem, the `rmfd` daemon needs to fully initialize the device. Among other things, the following steps will be performed:

- The daemon will make sure that a specific `/dev/cdc-wdm` character device will be used to get a specific `wwan` network connected.
- The QMI protocol will be **reset** so that any previously allocated service clients are disposed.
- The network interface will be explicitly configured to use **802.3** (ethernet) data format[9] as well as no QoS headers. 802.3 is the only mode that the `qmi-wwan` driver currently supports, so userspace needs to make sure that the correct mode is selected.
- The QMI protocol version information of the device will be preloaded in the QMI stack.
- The daemon will initialize several service-specific **QMI clients**, for the DMS (device management), NAS (network access), UIM (user identity module) and WMS (wireless messaging) services.

All the device initialization procedures are explicitly requested by the `rmfd` daemon using the interfaces provided by the `libqmi` library.

9 The default USB configuration in the Sierra Wireless MC7304 modem will expose 2 pairs of wwan interfaces, one configured by default to use 802.3 data format, and the other one raw IP packets (i.e. without ethernet headers).

3.4. SIM-PIN unlocking

The application using the RaveModemFactory system will need to make sure that the SIM-PIN is unlocked before trying to use the modem. The `librmf` library provides methods to both check if the unlock is necessary, and also to actually send the unlock code to the modem.

The SIM-PIN unlock operation will by default make sure that the SIM is fully initialized before returning the action result to the caller.

3.5. Network registration

Once SIM-PIN has been successfully unlocked, or if SIM-PIN lock was already disabled, the `rmfd` daemon will automatically request the modem to register in the **home network**. If the home network is not available, the modem will try to register in any roaming network where it's allowed to register. If the automatic registration procedure doesn't get the modem registered in 60s, the daemon will launch an explicit network scan operation to list the available networks, and once the scan is finished, it will restart the automatic registration in the home network.

3.6. Network connection

Once the modem is registered in a network, the user of the RaveModemFactory system will be able to request the modem to get connected to a specific **APN**. If the APN requires authentication, additional username and password parameters can also be given during the connection procedure.

If the connection call doesn't fail, the modem will be reported as connected in the QMI stack. The `rmfd` daemon will then bring the associated `wwan` interface **up** (using either `ifconfig` or `iproute2`) and will also request the **IP, routing and name resolution** setup via DHCP (using `dhclient`). The built-in DHCP server in the modem will take care of providing the expected settings to use.

Given that even the DNS resolution configuration is being managed, the rmfd daemon will backup the previous configuration[10] and will restore it whenever the QMI `wwan` interface gets disconnected.

Routing-wise, the `rmfd` daemon will take care of the **default route** once the QMI `wwan` interface gets connected, but will not re-select any other default route when disconnected.

3.7. Network disconnection

Network disconnection can be explicitly requested by the users of the RaveModemFactory system. When disconnected, the `wwan` network interface will be brought **down**, the DNS resolution configuration will be restored, and the call will be stopped in the QMI stack.

3.8. SMS messaging

The rmfd daemon will by default log to syslog all the SMS messages received by the mobile broadband modem, e.g.:

```
jan 12 11:27:03 athena rmfd[12805]: SMS [Timestamp: 150112112700+01] [From:
+34639335936] This is the SMS text content
```

10 `/etc/resolv.conf`

The `Timestamp:` field gives the SMS timestamp (YYMMDDhhmmss+utc offset), and the `From:` field indicates the sender number.

3.9. Statistics

The `rmfd` daemon keeps track of the ongoing connection statistics, as well as the monthly accumulated ones.

Each ongoing session is tracked in a stats log file[11], with a new record written every 10s containing the modem-reported statistics and other state values. Each record contains a list of TAB-separated fields, including:

- **Type of record:** `S` (start), `P` (partial) or `F` (final). When connection is launched, a `S` record is written, then `P` records every 10s until a last `F` record when disconnected in a controlled way.
- **Start timestamp:** Date and Time of when the connection was started. If the time can be retrieved from the modem itself it will be given in ISO8601 format, and otherwise it will include the Unix timestamp of the system.
- **Record timestamp:** Date and Time of when the record was written. If the time can be retrieved from the modem itself it will be given in ISO8601 format, and otherwise it will include the Unix timestamp of the system.
- **Duration:** Duration of the ongoing session, given in seconds.
- **Bytes RX:** Total bytes received, as measured by the modem itself.
- **Bytes TX:** Total bytes transmitted as measured by the modem itself.
- **Radio interface:** Radio interface being used in the connection (e.g. gsm, umts or lte).
- **RSSI:** Received Signal Strength Indication, given in dBm.
- **MCC:** Mobile country code of the serving system.
- **MNC:** Mobile network code of the serving system.
- **LAC:** Location area code of the serving system.
- **CID:** Cell ID of the serving system.

An full example of a single connection session would look like the following:

```
S  2015-03-19 09:49:56 2015-03-19 09:49:56  0    0      0      umts  -95   214  3   1140 10774738
P  2015-03-19 09:49:56 2015-03-19 09:50:06  10   22591  13816  umts  -95   214  3   1140 10774738
P  2015-03-19 09:49:56 2015-03-19 09:50:16  20   22845  14183  umts  -95   214  3   1140 10774738
P  2015-03-19 09:49:56 2015-03-19 09:50:26  30   22845  14183  umts  -95   214  3   1140 10774738
P  2015-03-19 09:49:56 2015-03-19 09:50:36  40   22845  14426  umts  -95   214  3   1140 10774738
P  2015-03-19 09:49:56 2015-03-19 09:50:46  50   23565  14502  umts  -104  214  3   1140 10774738
P  2015-03-19 09:49:56 2015-03-19 09:50:56  60   23565  14502  umts  -95   214  3   1140 10774738
P  2015-03-19 09:49:56 2015-03-19 09:51:06  70   23565  14615  umts  -95   214  3   1140 10774738
P  2015-03-19 09:49:56 2015-03-19 09:51:16  80   23565  14615  umts  -95   214  3   1140 10774738
P  2015-03-19 09:49:56 2015-03-19 09:51:26  90   23565  14615  umts  -95   214  3   1140 10774738
P  2015-03-19 09:49:56 2015-03-19 09:51:36  100  23565  14615  umts  -95   214  3   1140 10774738
P  2015-03-19 09:49:56 2015-03-19 09:51:46  110  23721  14735  umts  -95   214  3   1140 10774738
P  2015-03-19 09:49:56 2015-03-19 09:51:56  120  23721  14735  umts  -95   214  3   1140 10774738
F  2015-03-19 09:49:56 2015-03-19 09:52:06  130  23721  14848  umts  -95   214  3   1140 10774738
```

Once the connection has been fully disconnected, a new log will be written to syslog with the stats

11 `/var/log/rmfd.stats`

accumulated in the session:

```
mar 19 09:52:06 athena rmfd[12805]: Connection stats [from: 2015-03-19 09:49:56,
to: 2015-03-19 09:52:06, duration: 130s] [rx: 23721, tx: 14848] [access tech:
umts, rssi: -95dBm] [mcc: 214, mnc: 3, lac: 1140, cid: 10774738] [month 2015/3
rx: 23721, tx: 14848]
```

In addition to the last connection stats, the monthly accumulated RX/TX stats are also given.

If a connection is abruptly disconnected (e.g. system shuts down), the last valid record in the stats file is considered as the final record of the last connection, even if it's not a `F` (final) one. When `rmfd` restarts and detects that such situation happened, it will explicitly log to syslog the message corresponding to the last connection. In order to avoid additional logging to syslog of the same information it will also convert the last record to a `F` (final) one.

The stats log file is recreated every month. When `rmfd` detects that a new connection is happening in a month after the last one stored in the stats file, it will issue a new log to syslog before fully removing the stats log file contents:

```
mar 1 01:02:03 athena rmfd[999]: Month stats (2015/2) [rx: 586413, tx: 624682]
```

Assuming approximately 90 bytes per record, in the worst case of an ongoing connection 24/7 during the whole month, the size of the stats file may grow up to ~23MBytes:

```
90 * 6 * 60 * 24 * 31 = 24105600 ~23MB
```

4. librmf

4.1. Overview

The `librmf` library exposes the **`Modem` C++ namespace**, which includes the methods that allow remotely acting on the `rmfd` daemon. The user of the `librmf` library, doesn't need to know how the communication with the daemon is done.

All the methods implemented in the `librmf` library are **synchronous** and **blocking**. It is expected that if asynchronity is desired, the user application will take care of e.g. launching these methods in separate threads.

Errors in the `librmf` library methods will throw `std::runtime_error` **exceptions**, including a text string explaining the error that happened.

All the method executions in the librmf library are **stateless** and independent to each other. Multiple processes, or even multiple threads in the same process, can run actions in parallel using `librmf`.

By default the `librmf` library assumes that the communication with the `rmfd` daemon will be local through the standard **`UNIX` socket**. If the `rmfd` daemon is running in a different system and listening in a TCP port on a remote IP address, the client can instruct the library to default to using **TCP/IP** instead.

4.2. Usage

The `librmf` library can be used by including the header file in the sources where you want to use the API methods:

```
#include <rmf-operations.h>
```

And also (assuming it was installed in `/usr`) adding the following arguments during the preprocessing and linking operations:

```
-I/usr/include/librmf        // preprocessing
-lrmf                        // linking
```

A full simple example of usage is included later in section 4.4.

4.3. API

The following sections show the application programming interface exposed by `librmf`.

4.3.1. Library initialization API

```
bool SetTargetRemote (const std::string address,
                      uint16_t          port);
bool SetTargetLocal  (void);
```

Before any other operation is done, the client can specify where the rmfd daemon is listening. By default the daemon is expected in the local `UNIX` socket, so this operation is only strictly needed if the daemon is listening on a remote (or local) TCP port.

The target setting may be changed only if there are no other API commands running.

4.3.2. Device availability API

```
bool IsModemAvailable (void);
```

The user of the `librmf` library does not need to know which kind of modem is connected to the system, and not even which are the interfaces the modem is exposing. All that information is automatically managed by the `rmfd` daemon, which will just tell whether a suitable modem is available or not.

If more than one modem is available, or if a modem exposes multiple QMI interfaces, the first configured one will be used. If the modem is available but not ready to work (e.g. SIM card is missing), it will still be flagged as available, but the operations performed with it will end up failing.

4.3.3. Device identification API

```
std::string GetManufacturer     (void);
std::string GetModel            (void);
std::string GetSoftwareRevision (void);
std::string GetHardwareRevision (void);
std::string GetImei             (void);
```

The RaveModemFactory system allows to query the modem for several device identification strings, including the IMEI[12].

The device identification strings will be retrievable even if the SIM-PIN lock is enabled and locked.

4.3.4. SIM identification API

```
struct PlmnInfo {
    uint16_t mcc;
    uint16_t mnc;
    bool     gsm;
    bool     umts;
    bool     lte;
};

std::string GetImsi    (void);
std::string GetIccid   (void);
void        GetSimInfo (uint16_t                    &operatorMcc,
                        uint16_t                    &operatorMnc,
                        std::vector<struct PlmnInfo> &plmns);
```

The RaveModemFactory system allows to query the modem for several SIM identification strings, including the IMSI[13] and the ICCID[14].

The `GetSimInfo` method returns not only the MCC/MNC[15] of the network operator which issued the SIM card (as per the `EFad`[16] SIM file), but also a list of other allowed operators with their access technologies (as per the `EFoplmnwact`[17] SIM file, if available).

Note that the previous methods may not return valid information until the SIM-PIN has been totally unlocked.

12 http://en.wikipedia.org/wiki/International_Mobile_Station_Equipment_Identity
13 http://en.wikipedia.org/wiki/International_mobile_subscriber_identity
14 http://en.wikipedia.org/wiki/Subscriber_identity_module#ICCID
15 http://en.wikipedia.org/wiki/Mobile_country_code
16 Administrative Data, 3GPP TS 31.121
17 Operator Controlled PLMN Selector with Access Technology, 3GPP TS 31.121

4.3.5. SIM-PIN management API

```
bool IsSimLocked (void);
void Unlock      (const std::string pin);
void EnablePin   (bool              enable,
                  const std::string pin);
void ChangePin   (const std::string pin,
                  const std::string newPin);
```

In addition to checking the SIM-PIN status and unlocking it if necessary, the RaveModemFactory system also allows to enable, disable, and change the SIM-PIN lock code.

The `IsSimLocked()` method will return `false` if either the SIM-PIN is fully disabled or if it is enabled but already unlocked through `Unlock()`.

Working with SIM-PUK is not directly available through the `librmf` interfaces. If for any reason the SIM gets locked asking for PUK, the `qmicli` tool can be used to unlock it using `--dms-uim-unblock-pin`.

4.3.6. Power management API

```
enum PowerStatus {
    Full,
    Low
};

PowerStatus GetPowerStatus (void);
void        SetPowerStatus (PowerStatus powerStatus);
void        PowerCycle     (void);
```

The RaveModemFactory system allows to put the modem into low-power mode (i.e. RF off), and back to full-power mode (i.e. RF on). Obviously, the modem needs to be in full-power mode before it can be registered in the operator network.

If the modem is initially detected in low-power mode, the `rmfd` daemon will not automatically bring it into full-power mode. Additionally, re-setting the `Full` power status (even if it already is in that status) will trigger a new network scan if the modem is not yet registered to the operator network.

A full modem power cycle can also be requested using the `PowerCycle()` method. In this case, the `rmfd` daemon will reply as soon as the request has been accepted; i.e. it will not wait for the full reboot sequence to finish before replying.

4.3.7. Network registration API

```
enum RegistrationStatus {
    Idle,
    Searching,
    Home,
    Roaming,
    Scanning
};

RegistrationStatus GetRegistrationStatus  (std::string   &operatorDescription,
                                           uint16_t      &operatorMcc,
                                           uint16_t      &operatorMnc,
                                           uint16_t      &lac,
                                           uint32_t      &cid);
void               SetRegistrationTimeout (uint32_t       timeout);
uint32_t           GetRegistrationTimeout (void);
```

The RaveModemFactory has its own automatic logic to get registered in the mobile network. Therefore, librmf only allows checking the registration status, or querying and modifying the registration timeout before the built-in network scan request is launched.

The returned registration info includes not only the MCC/MNC[18] of the service provider network, but also the location area code[19] and cell id[20] where the modem is registered, which can be used to locate the position of the device with low accuracy.

4.3.8. Network status API

```
enum RadioInterface {
    Gsm,
    Umts,
    Lte
};
struct RadioPowerInfo {
    RadioInterface radioInterface;
    bool           inTraffic;
    double         txPower;
    bool           rx0RadioTuned;
    double         rx0Power;
    bool           rx1RadioTuned;
    double         rx1Power;
};

struct RadioSignalInfo {
    RadioInterface radioInterface;
    int32_t        rssi;
    uint32_t       quality;
};

std::vector<RadioPowerInfo>  GetPowerInfo  (void);
std::vector<RadioSignalInfo> GetSignalInfo (void);
```

The `GetPowerInfo()` method returns an array of structures which specify, per radio interface, whether the TX and RX chains are in traffic and what the power measurement is in each (given in dBm, only if in traffic).

The `GetSignalInfo()` method returns an array of structures which specify, per radio interface, the quality of the network link. The quality is given both as the measured RSSI[21] (in dBm, where -125dBm or lower indicates no signal) and as a percentage.

4.3.9. Network connection API

```
enum ConnectionStatus {
    Disconnected,
    Disconnecting,
    Connecting,
    Connected
};

ConnectionStatus GetConnectionStatus (void)
void             Connect             (const std::string apn,
```

18 http://en.wikipedia.org/wiki/Mobile_country_code
19 http://en.wikipedia.org/wiki/Mobility_management#Location_area
20 http://en.wikipedia.org/wiki/Cell_ID
21 http://en.wikipedia.org/wiki/Received_signal_strength_indication

```
                                    const std::string user,
                                    const std::string password)
void            Disconnect         (void);
std:string      GetDataPort        (void);
```

The user of the librmf library can request connection to the network with the `Connect()` method, just passing the APN, user and password strings. If no user or password are needed, they should be given as empty strings.

The `Connect()` method will return only after all the connection steps have been performed, including the network interface and routing setup.

The `Disconnect()` method allows the user to request the disconnection from the network. In the same way, the method will only return once all the disconnection steps have been performed.

The `Disconnecting` and `Connecting` states returned by the `GetConnectionStatus()` method are only applicable when the method is run from another thread or process than the one which requested the actual connection or disconnection.

The `GetDataPort()` method will return the name of the network interface being managed by `rmfd`, useful for the case where multiple network interfaces are exposed by a single modem and only one is used.

4.3.10. Network connection statistics API

```
bool GetConnectionStats (uint32_t &txPacketsOk,
                         uint32_t &rxPacketsOk,
                         uint32_t &txPacketsError,
                         uint32_t &rxPacketsError,
                         uint32_t &txPacketsOverflow,
                         uint32_t &rxPacketsOverflow,
                         uint64_t &txBytesOk,
                         uint64_t &rxBytesOk);
```

The RaveModemFactory allows to query which are the number of received and transmitted packets and bytes in the ongoing connection, if any.

4.4. Example

```
/*
 * Small librmf client example
 *
 * Assuming RMF was installed in /usr, compile with:
 *   $> g++ -o example -I /usr/include/librmf -lrmf example.cpp
 */

#include <iostream>
#include <rmf-operations.h>

int
main (int argc, char **argv)
{
    std::string manufacturer;

    try {
        manufacturer = Modem::GetManufacturer ();
    } catch (std::exception const& e) {
        std::cout << "Exception: " << e.what() << std::endl;
        return -1;
    }

    std::cout << "Manufacturer: " << manufacturer << std::endl;
    return 0;
}
```

5. rmfcli

The command line interface provided in the RaveModemFactory system allows to test every action that the `librmf` library exposes.

Note: `rmfcli` is just a test tool.
The suggested way to use RMF is to write a custom connection manager that uses `librmf` directly.

The tool can be run even if there is another user application using the `librmf` interfaces.

The following lines show the output of running `rmfcli --help`:

```
Usage: rmfcli <option>

Options:
       -y, --target-address="ip"
       -Y, --target-port="port"

Actions:
       -f, --get-manufacturer
       -d, --get-model
       -j, --get-software-revision
       -k, --get-hardware-revision
       -e, --get-imei
       -i, --get-imsi
       -o, --get-iccid
       -z, --get-sim-info
       -L, --is-locked
       -U, --unlock="pin"
       -E, --enable-pin="pin"
       -G, --disable-pin="pin"
       -F, --change-pin="pin newpin"
       -p, --get-power-status
       -P, --set-power-status="[Full|Low]"
       -Z, --power-cycle
       -a, --get-power-info
       -s, --get-signal-info
       -r, --get-registration-status
       -t, --get-registration-timeout
       -T, --set-registration-timeout="timeout"
       -c, --get-connection-status
       -x, --get-connection-stats
       -C, --connect="apn user password"
       -D, --disconnect
       -A, --is-available

Common actions:
       -h, --help
       -v, --version
```