



Connectivity documentation

1 Contents

2	Writing ConnMan plugins	2
3	Customs ConnMan Session policies	2
4	Management of ConnMan Sessions	2
5	WiFi radio start up behavior on ConnMan	3
6	Supporting new data modems in oFono	3
7	Writing new Telepathy Connection Managers	4
8	Looking inside the telepathy-rakia code	5
9	Writing new Folks backends	9

10 Writing ConnMan plugins

11 The plugin documentation in ConnMan was improved and submitted upstream.
12 The documentation about writing plugins can be found on ConnMan sources in
13 the following files: *doc/plugin-api.txt*, *src/device.c* and *src/network.c*. Example
14 plugins are *plugins/bluetooth.c* *plugins/wifi.c*, *plugins/ofono.c*, among others.

15 Customs ConnMan Session policies

16 The documentation to create Session policies files for specifics users and/or
17 groups can be found in ConnMan sources *doc/session-policy-format.txt*. The
18 policies files shall be placed in `STORAGEDIR/session_policy_local` directory, where
19 `STORAGEDIR` by default points to `/var/lib/connman`. ConnMan can recognize
20 changes to this directory during runtime and update Session policies accordingly.

21 Management of ConnMan Sessions

22 ConnMan provides a extensive API to manage the creation, configuration and
23 removal of a session, *doc/manager-api.txt* details how to create and destroy a Ses-
24 sion through the `CreateSession()` and `DestroySession()` methods. *doc/session-*
25 *api.txt* details how to use a Session. Through this API an application can ask
26 ConnMan to Connect/Disconnect a Session or change its settings. The Settings
27 can also be changed by writing policies files as described in the previous topic.

28 The application requesting a Session needs to implement a Notification API to
29 receive updates in the Session settings, such as when a Session becomes online.
30 This is done via the `Update()` method.

31 See also *doc/session-overview.txt*.

32 The difference between using the Session API and the policy files in
33 `/var/lib/connman` is that policy files can set policies to many sessions at the
34 same time, based on user/group ID or SELINUX rules while Session API only
35 changes one session at a time.

36 WiFi radio start up behavior on ConnMan

37 At the very first run ConnMan has the WiFi radio disabled by default, however
38 sometimes it is important to have the radio enabled even in the first ConnMan
39 run. To achieve this behavior ConnMan can be configured to enable the radio
40 on it first run.

41 The file `STORAGEDIR/settings`, where `STORAGEDIR` by default points to
42 `/var/lib/connman`, shall be edited, or even created, to have the following con-
43 tent:

```
44 [WiFi]  
45  
46 Enable=true
```

47 This configuration will tell ConnMan at start up to enable the WiFi radio.

48 Supporting new data modems in oFono

49 oFono has a great support for most of the modems out there in the market,
50 however some new modem may not work out-of-the-box, in this case we need to
51 fix oFono to recognize and handle the new modem properly. There are a couple
52 of different causes why a modem does not work with oFono. In this section we
53 will detail them and show how oFono can be fixed.

- 54 • Modem match failure: if the `udevng` plugin in oFono fails to match the
55 new modem its code needs to be fixed to recognize the new modem. This
56 kind of failure can be recognized by looking at the debug output of the
57 `udevng` plugin (debug output is enabled when running `ofonod` with the
58 `-d` option). If `udevng` doesn't say anything about the new modem then it
59 needs proper code to handle it. You can find an example on how to edit
60 `plugins/udevng.c` to support a new modem in [oFono git](https://git.kernel.org/cgit/network/ofono/ofono.git/commit/?id=d1ac1ba3d474e56593ac3207d335a4de3d1f4a1d)¹. The oFono git
61 history has many examples of patches to add support to new modems in
62 `plugins/udevng.c`
- 63 • Some other modems does not implement the specifications properly and
64 thus oFono needs to implement 'quirks' to have these modems working
65 properly. Many examples of fixes can be found on oFono git:

- 66 – <https://git.kernel.org/cgit/network/ofono/ofono.git/commit/?id=d1ac1ba3d474e56593ac3207d335a4de3d1f4a1d>
- 68 – <https://git.kernel.org/cgit/network/ofono/ofono.git/commit/?id=535ff69deddda292c7047620dc11336dfb480a0d>

70 It is difficult to foresee the problems that can happen when trying a new modem
71 due to the extensive number of commands and specifications oFono implements.

¹<https://git.kernel.org/cgit/network/ofono/ofono.git/commit/?id=4cabdedafdc241706e342720a20bdfc3828dfadf>

72 Asking the [oFono community](https://01.org/ofono)² could be very helpful to solve any issue with a
73 new modem.

74 Writing new Telepathy Connection Managers

75 New connection managers are implemented as separated component and have
76 their own process. Telepathy defines the [D-Bus interfaces](http://telepathy.freedesktop.org/doc/book/sect.basics.interfaces.html)³ that each Connection
77 Manager (CM) needs to implement. This is known as the Telepathy Specifica-
78 tion.

79 The Connection Managers need to expose a bus name in D-Bus that begins
80 with *org.freedesktop.Telepathy.ConnectionManager*, for example, the telepathy-
81 gabble CM, has the *org.freedesktop.Telepathy.ConnectionManager.gabble* bus
82 name to provide its XMPP protocol interfaces.

83 A client that wants to talk to the available Connection Managers in the D-Bus
84 Session bus needs to call D-Bus *ListActivatableNames* method and search for
85 names with the returned prefix.

86 The most important Interfaces that a Connection Manager needs to implement
87 are *ConnectionManager*, *Connection* and *Channel*. The *ConnectionManager*
88 handles creation and destruction of *Connection* object. A *Connection* object
89 represents a connected protocol session, such as a XMPP session. Within a
90 *Connection* many *Channel* objects can be created; they are used for communi-
91 cation between the application and the server providing the protocol service.
92 A *Channel* can represent many different types of communications such as files
93 transfers, incoming and outgoing messages, contact search, etc.

94 Another important concept is the [Handle](http://telepathy.freedesktop.org/doc/book/sect.basics.handles.html)⁴. It is basically a numeric ID to
95 represent various protocol resources, such as contacts, chatrooms, contact lists
96 and user-defined groups.

97 The [Telepathy Developer's Manual](http://telepathy.freedesktop.org/doc/book/)⁵ details how to use the Telepathy API and
98 thus gives many suggestions of how those should be implemented by a new
99 Connection Manager.

100 Studying the code of existing Connection Managers is informative when imple-
101 menting a new one. Two good examples are [telepathy-gabble](https://github.com/TelepathyIM/telepathy-gabble)⁶ for the XMPP
102 protocol or [telepathy-rakia](https://github.com/TelepathyIM/telepathy-rakia)⁷ for the SIP implementation.

103 Those Connection Managers use [Telepathy-GLib](https://github.com/TelepathyIM/telepathy-glib)⁸ as a framework to implement
104 the Telepathy Specification. The Telepathy-GLib repository has [a few exam-](#)

²<https://01.org/ofono>

³<http://telepathy.freedesktop.org/spec/>

⁴<http://telepathy.freedesktop.org/doc/book/sect.basics.handles.html>

⁵<http://telepathy.freedesktop.org/doc/book/>

⁶<https://github.com/TelepathyIM/telepathy-gabble>

⁷<https://github.com/TelepathyIM/telepathy-rakia>

⁸<https://github.com/TelepathyIM/telepathy-glib>

105 [ples](#)⁹ of its usage.

106 It is strongly recommend to use Telepathy-GLib when implementing any new
107 connection manager. The Telepathy-GLib service-side API is only available in
108 C, but can also be access from other languages that can embed C, such as C++.
109 This library is [fully documented](#)¹⁰.

110 Looking inside the telepathy-rakia code

111 To start, a small design document can be found at *docs/design.txt* in telepathy-
112 rakia sources. However, some parts of it are outdated.

113 Source files

- 114 • *src/telepathy-rakia.c*: this is the starting point of telepathy-rakia as it
115 instantiates its *ConnectionManager*.
- 116 • *src/sip-connection-manager.[ch]*: defines the *ConnectionManagerClass*
117 and requests the creation of a *Protocol* of type *TpBaseProtocol*.
- 118 • *src/protocol.[ch]*: defines the *RakiaProtocolClass* which creates the *Tp*-
119 *BaseProtocol* object. The protocol is responsible for starting new *Connec*-
120 *tions*. The request arrives via D-Bus and arrives here through Telepathy-
121 GLib.
- 122 • *src/sip-connection.c*: defines the *RakiaConnectionClass* which inherits
123 from *RakiaBaseConnectionClass*. The latter inherits from *TpBaseCon*-
124 *nectionClass*.
- 125 • *src/sip-connection-helpers.[ch]*: helper routines used by *RakiaConnection*
- 126 • *src/sip-connection-private.h*: private structures for *RakiaConnection*
- 127 • *src/write-mgr-file.c*: utility to produce manager files
- 128 • *rakia/base-connection.[ch]*: base class for *RakiaConnectionClass*. It imple-
129 ments its parent, *RakiaBaseConnectionClass*
- 130 • *rakia/base-connection-sofia.[ch]*: Implements a callback to handle events
131 from the SIP stack.
- 132 • *rakia/text-manager.[ch]*: defines *RakiaTextManagerClass*, to manage the
133 *RakiaTextChannel*.
- 134 • *rakia/text-channel.[ch]*: defines *RakiaTextChannelClass*. This is a Telepa-
135 thy *Channel*.
- 136 • *rakia/media-manager.[ch]*: defines *RakiaMediaManagerClass*. Handles
137 the *RakiaSipSession*.

⁹<https://github.com/TelepathyIM/telepathy-glib/blob/master/examples/README>

¹⁰<http://telepathy.freedesktop.org/doc/telepathy-glib/>

- 138 • *rakia/sip-session.[ch]*: defines *RakiaSipSessionClass*; it relates directly to
139 the definition of Session in the SIP specification.
- 140 • *rakia/call-channel.[ch]*: defines *RakiaCallChannelClass*. The object is cre-
141 ated when an incoming calls arrives or an outgoing call is placed. A
142 *RakiaCallChannel* belongs to one *RakiaSipSession*.
- 143 • *rakia/sip-media.[ch]*: defines *RakiaSipMediaClass*. It is created immedi-
144 ately after a *RakiaCallChannel* is created. Can represent audio or video
145 content.
- 146 • *rakia/call-content.[ch]*: defines *RakiaCallContentClass*. The object is cre-
147 ated for each new medium added. It relates directly to the *Content* defini-
148 tion in the Telepathy specification. It could be an audio or video *Content*,
149 it is matched one-to-one with a *RakiaSipMedia* object.
- 150 • *rakia/call-stream.[ch]*: defines the *RakiaCallStreamClass*. It could be an
151 audio or video object. The object is created by *RakiaCallContent*.
- 152 • *rakia/codec-param-formats.[ch]*: helper to setting codecs parameters.
- 153 • *rakia/connection-aliasing.[ch]*: defines function for aliasing *Connections*.
- 154 • *rakia/debug.[ch]*: debug helpers
- 155 • *rakia/event-target.[ch]*: helper to listen for events for a NUA handle (see
156 NUA definition in sofia-sip documentation).
- 157 • *rakia/handles.[ch]*: helpers for *Handles*.
- 158 • *rakia/sofia-decls.h*: some extra declaration
- 159 • *rakia/util.[ch]*: utility functions.

160 **sofia-sip** [sofia-sip](http://sofia-sip.sourceforge.net/)¹¹ is a User-Agent library that implements the SIP protocol
161 as described in IETF RFC 3261. It can be used for VoIP, IM, and many other
162 real-time and person-to-person communication services. telepathy-rakia makes
163 use of sofia-sip to implement SIP support into telepathy. sofia-sip has [good](http://sofia-sip.sourceforge.net/refdocs/nua/)
164 [documentation](http://sofia-sip.sourceforge.net/refdocs/nua/)¹² on all concepts, events and APIs.

165 **Connection Manager and creating connections** *src/telepathy-rakia.c* is
166 the starting point of this Telepathy SIP service. Its *main()* function does some
167 of the initial setup, including D-Bus and *Logging* and calls Telepathy-GLib's
168 *tp_run_connection_manager()* method. The callback passed to this method
169 gets called and constructs a new Telepathy *ConnectionManager GObject*. The
170 Connection Manager Factory is at *src/sip-connection-manager.c*.

171 Once the Connection Manager Object construction is finalized, the creation of a
172 SIP Protocol Object is triggered inside *rakia_connection_manager_constructed()*

¹¹<http://sofia-sip.sourceforge.net/>

¹²<http://sofia-sip.sourceforge.net/refdocs/nua/>

173 by calling *rakia_protocol_new()*. This function is defined in *src/protocol.c*.
 174 It creates a Protocol Object and adds the necessary infrastructure that a
 175 Connection Manager needs to manage the Protocol. In the Class Factory it
 176 is possible to see which methods are defined by this Class by looking at the
 177 *TpBaseProtocolClass* *base_class* var:

```
178 base_class->get_parameters = get_parameters;
179 base_class->new_connection = new_connection;
180 base_class->normalize_contact = normalize_contact;
181 base_class->identify_account = identify_account;
182 base_class->get_interfaces = get_interfaces;
183 base_class->get_connection_details = get_connection_details;
184 base_class->dup_authentication_types = dup_authentication_types;
```

185 Documentation on each method of this class can be found in the Telepathy-
 186 GLib documentation for [TpBaseConnectionManager](#)¹³ and [TpBaseProtocol](#)¹⁴. The *Protocol* is bound to *ConnectionManager* through the method
 187 *tp_base_connection_manager_add_protocol()*.
 188

189 The *new_connection()* method defined there is used to create a new Telepathy
 190 *Connection* when the *NewConnection()* method on *org.freedesktop.Telepathy.ConnectionManager.rakia*
 191 is called.

192 The Telepathy *Connection* object is of type *RakiaConnection*, which inherits
 193 from *RakiaBaseConnection*, which in turn inherits from *TpBaseConnection*. The
 194 methods used by *RakiaConnection* can be seen at the *RakiaConnectionClass*
 195 and *RakiaBaseConnectionClass* initializations. They are defined at *src/sip-*
 196 *connection.c* for the *RakiaBaseConnectionClass*:

```
197 sip_class->create_handle = rakia_connection_create_nua_handle;
198 sip_class->add_auth_handler =
199 rakia_connection_add_auth_handler;
```

200 and for the *TpBaseConnectionClass*:

```
201 base_class->create_handle_repos = rakia_create_handle_repos;
202 base_class->get_unique_connection_name = rakia_connection_unique_name;
203 base_class->create_channel_managers = rakia_connection_create_channel_managers;
204 base_class->create_channel_factories = NULL;
205 base_class->disconnected = rakia_connection_disconnected;
206 base_class->start_connecting = rakia_connection_start_connecting;
207 base_class->shut_down = rakia_connection_shut_down;
208 base_class->interfaces_always_present =
209 interfaces_always_present;
```

210 During the *TpBaseConnection* object construction the *create_channel_managers*
 211 method is called. A *Channel* is an entity provided by a *Connection* to allow the

¹³<http://telepathy.freedesktop.org/doc/telepathy-glib/TpBaseConnectionManager.html>

¹⁴<http://telepathy.freedesktop.org/doc/telepathy-glib/telepathy-glib-base-protocol.html>

communication between the local *ConnectionManager* and the remote server providing the service. A *Channel* can represent an incoming or outgoing IM message, a file transfer, a video call, etc. Many *Channels* can exist at a given time.

Channels and Calls telepathy-rakia has two types of *Channels*: *Text* and *Call*. For *TextChannels* a *RakiaTextManager* object is created. It inherits from *TpChannelManager*. *TpChannelManager* is a generic type used by all types of *Channels*. See *rakia/text-manager.c* for the *RakiaTextManagerClass* definitions. When constructed, in *rakia_text_manager_constructed()*, the object sets the *connection_status_changed_cb* callback to get notified about *Connection* status changes. If the *Connection* status changes to *Connected*, the callback is activated and the code sets yet another callback, *rakia_nua_i_message_cb*. This callback is connected to nua-event from sofia-sip. This callback is responsible for managing an incoming message request from the remote server.

The callback then handles the message it receives through the *Connection* using the sofia-sip library. At the end of the function the following code can be found:

```
channel = rakia_text_manager_lookup_channel (fac, handle);
if (!channel)
    channel = rakia_text_manager_new_channel (fac, handle, handle, NULL);
rakia_text_channel_receive (channel, sip, handle, text, len);
```

The *RakiaTextManager* tries to figure if an existing *Channel* for this message already exists, or if a new one needs to be created. Once the channel is found or created, *RakiaTextManager* is notified of the received message through *rakia_text_channel_receive()* which creates a *TpMessage* to wrap the received message.

A similar process happens with the similar *RakiaMediaManager* which handles SIP *Sessions* and *Call Channels*. The callback registered by *RakiaMediaManager* is *rakia_nua_i_invite_cb()*, in *rakia/media-manager.c*, it then can get notified of incoming invites to create a SIP *Session*. Once the callback is activated, which means when an incoming request to create a SIP *Session* arrives, a new *RakiaSipSession* is created. Outgoing requests to create a SIP session *RakiaSipSession* are initiated on the telepathy-rakia side through the exposed D-Bus interface. The request comes from the *TpChannelManager* object and is created by *rakia_media_manager_requestotron()* in the end of its call chain:

```
static void
channel_manager_iface_init (gpointer g_iface, gpointer iface_data)
{
    TpChannelManagerInterface *iface = g_iface;
    iface->foreach_channel = rakia_media_manager_foreach_channel;
    iface->type_foreach_channel_class = rakia_media_manager_type_foreach_channel_class;
    iface->request_channel = rakia_media_manager_request_channel;
    iface->create_channel = rakia_media_manager_create_channel;
```



```

254         iface->ensure_channel = rakia_media_manager_ensure_channel;
255     }

```

Here in *channel_manager_iface_init()*, telepathy-rakia sets which method it wants to be called when the [D-Bus methods](#)¹⁵ exposed by Telepathy-GLib are called. These functions handle *Channel* creation; however, they must first create a SIP *Session* before creating the *Channel* itself. The *RakiaSipSession* object will handle the *Channels* between the remote server and telepathy-rakia.

In the incoming path besides of creating a new SIP session the *rakia_nua_i_invite_cb* callback also sets a new callback *incoming_call_cb*, that as its name says gets called when a new call arrives.

CallChannels, implemented as *RakiaCallChannel* in telepathy-rakia, are then created once this callback is activated or, for outgoing call channels requests, just after the *RakiaSipSession* is created. See the calls to *new_call_channel()* inside *rakia/media-manager.c* for more details.

If *RakiaCallChannel* constructed was requested by the local user up to two new media streams would be created and added to it; the media can be audio or video. The media streams, known as a *RakiaSipMedia* object, is either created by the *CallChannel* constructed method if [InitialAudio](#)¹⁶ or [InitialVideo](#)¹⁷ is passed or by a later call to *AddContent()* on the D-Bus interface *org.freedesktop.Telepathy.Channel.Type.Call1*.

The creation of a *Content* object adds a “m=” line in the SDP in the SIP message body. Refer to the RFC 3261 specification.

The last important concept is a *CallStream*, implemented here as *RakiaCallStream*. A *CallStream* represents either a video or an audio stream to one specific remote participant, and is created through *rakia_call_content_add_stream()* every time a new *Content* object is created. In telepathy-rakia each *Content* object only has only one *Stream* because only one-to-one calls are supported.

Writing new Folks backends

The [Folks documentation](#)¹⁸ on backends is fairly extensive and can help quite a lot when writing a new backend. Each backend should provide a subclass of [Folks.Backend](#)¹⁹.

The same documentation can be found in the sources in the file *folks/backend.vala*. The evolution-data-server (EDS) backend will be used as example here due to its extensive documentation. The EDS subclass for *Folks.Backend* is defined in *backend/eds/eds-backend.vala* in the sources.

¹⁵http://telepathy.freedesktop.org/spec/Connection_Interface_Requests.html

¹⁶http://telepathy.freedesktop.org/spec/Channel_Type_Call.html#Property:InitialAudio

¹⁷http://telepathy.freedesktop.org/spec/Channel_Type_Call.html#Property:InitialVideo

¹⁸<https://wiki.gnome.org/Projects/Folks>

¹⁹<http://telepathy.freedesktop.org/doc/folks/vala/Folks.Backend.html>

289 A backend also needs to implement the [Folks.Persona](#)²⁰ and [Folks.PersonaStore](#)²¹
290 subclassess. For EDS those are [Edsf.Persona](#)²² and [Edsf.PersonaStore](#)²³, which
291 can also be seen in the sources in *backends/eds/lib/edsf-persona.vala* and
292 *backends/eds/lib/edsf-persona-store.vala*, respectively.

293 *Persona* is the representation of a single contact in a given backend, they are
294 stored by a *PersonaStore*. One backend may have many *PersonaStores* if they
295 happen to have different sources of contacts. For instance, each EDS address
296 book would have an associated *PersonaStore* to it. *Personas* from different
297 *Backends* that represent the same physical person are aggregated together by
298 Folks core as a [Individual](#)²⁴.

299 The Telepathy backend also serves as a good example. As the EDS backend, it
300 is well-implemented and documented.

²⁰<http://telepathy.freedesktop.org/doc/folks/vala/Folks.Persona.html>

²¹<http://telepathy.freedesktop.org/doc/folks/vala/Folks.PersonaStore.html>

²²<http://telepathy.freedesktop.org/doc/folks-eds/vala/Edsf.Persona.html>

²³<http://telepathy.freedesktop.org/doc/folks-eds/vala/Edsf.PersonaStore.html>

²⁴<http://telepathy.freedesktop.org/doc/folks/vala/Folks.Individual.html>