APERTIS

Connectivity documentation

Contents

Writing ConnMan plugins

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

Customs ConnMan Session policies

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

Management of ConnMan Sessions

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

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

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

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

WiFi radio start up behavior on ConnMan

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

The file STORAGEDIR/settings, where STORAGEDIR by default points to /var/lib/connman, shall be edited, or even created, to have the following content:

```
[WiFi]

Enable=true
```

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

Supporting new data modems in oFono

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

- Modem match failure: if the udevng plugin in oFono fails to match the new modem its code needs to be fixed to recognize the new modem. This kind of failure can be recognized by looking at the debug output of the udevng plugin (debug output is enabled when running ofonod with the '-d' option). If udevng doesn't say anything about the new modem then it needs proper code to handle it. You can find a example on how to edit plugins/udevng.c to support a new modem in oFono git[1]. The oFono git history has many examples of patches to add support to new modems in plugins/udevng.c

- Some other modems does not implement the specifications properly and thus oFono needs to implement 'quirks' to have these modems working properly. Many examples of fixes can be found on oFono git:
  - https://git.kernel.org/cgit/network/ofono/ofono.git/commit/?id=d1ac1ba3d474e56593ac3207d335a4de3d1f4a1d
  - https://git.kernel.org/cgit/network/ofono/ofono.git/commit/?id=535ff69deddda292c7047620dc11336dfb480a0d

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

[1] https://git.kernel.org/cgit/network/ofono/ofono.git/commit/?id=4cabdedafdc241706e342720a20bdfe3828dfadf

Asking the oFono community[2] could be very helpful to solve any issue with a new modem.

Writing new Telepathy Connection Managers

New connection managers are implemented as separated component and have their own process. Telepathy defines the D-Bus interfaces[3] that each Connection Manager (CM) needs to implement. This is known as the Telepathy Specification.

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

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

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

Another important concept is the Handle[4]. It is basically a numeric ID to represent various protocol resources, such as contacts, chatrooms, contact lists and user-defined groups.

The Telepathy Developer's Manual[5] details how to use the Telepathy API and thus gives many suggestions of how those should be implemented by a new Connection Manager.

Studying the code of existing Connection Managers is informative when implementing a new one. Two good examples are telepathy-gabble[6] for the XMPP protocol or telepathy-rakia[7] for the SIP implementation.

Those Connection Managers use Telepathy-GLib[8] as a framework to implement 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] http://cgit.freedesktop.org/telepathy/telepathy-gabble/
[7] http://cgit.freedesktop.org/telepathy/telepathy-rakia/
[8] http://cgit.freedesktop.org/telepathy/telepathy-glib/

ples[9] of its usage.

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

Looking inside the telepathy-rakia code

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

Source files

- *src/telepathy-rakia.c*: this is the starting point of telepathy-rakia as it instantiates its *ConnectionManager*.
- *src/sip-connection-manager.[ch]*: defines the *ConnectionManager*Class and requests the creation of a *Protocol* of type *TpBaseProtocol*.
- *src/protocol.[ch]*: defines the *RakiaProtocolC*lass which creates the *TpBaseProtocol* object. The protocol is responsible for starting new *Connections*. The request arrives via D-Bus and arrives here through Telepathy-GLib.
- *src/sip-connection.c*: defines the *RakiaConnectionClass* which inherits from *RakiaBaseConnectionClass*. The latter inherits from *TpBaseConnectionClass*.
- *src/sip-connection-helpers.[ch]*: helper routines used by *RakiaConnection*
- *src/sip-connection-private.h*: private structures for *RakiaConnection*
- *src/write-mgr-file.c*: utility to produce manager files
- *rakia/base-connection.[ch]*: base class for *RakiaConnectionClass*. It implements its parent, *RakiaBaseConnectionClass*
- *rakia/base-connection-sofia.[ch]*: Implements a callback to handle events from the SIP stack.
- *rakia/text-manager.[ch]*: defines *RakiaTextManagerClass*, to manage the *RakiaTextChannel*.
- *rakia/text-channel.[ch]*: defines *RakiaTextChannelClass*. This is a Telepathy *Channel*.
- *rakia/media-manager.[ch]*: defines *RakiaMediaManagerClass*. Handles the *RakiaSipSession*.

[9]http://cgit.freedesktop.org/telepathy/telepathy-glib/tree/examples/README
[10]http://telepathy.freedesktop.org/doc/telepathy-glib/

- *rakia/sip-session.[ch]*: defines *RakiaSipSessionClass*; it relates directly to the definition of Session in the SIP specifcation.
- *rakia/call-channel.[ch]*: defines *RakiaCallChannelClass*. The object is created when an incoming calls arrives or an outgoing call is placed. A *RakiaCallChannel* belongs to one *RakiaSipSession*.
- *rakia/sip-media.[ch]*: defines *RakiaSipMediaClass*. It is created immediately after a *RakiaCallChannel* is created. Can represent audio or video content.
- *rakia/call-content.[ch]*: defines *RakiaCallContentClass*. The object is created for each new medium added. It relates directly to the *Content* definition in the Telepathy specification. It could be an audio or video *Content*, it is matched one-to-one with a *RakiaSipMedia* object.
- *rakia/call-stream.[ch]*: defines the *RakiaCallStreamClass*. It could be an audio or video object. The object is created by *RakiaCallContent*.
- *rakia/codec-param-formats.[ch]*: helper to setting codecs parameters.
- *rakia/connection-aliasing.[ch]*: defines function for aliasing *Connection*s.
- *rakia/debug.[ch]*: debug helpers
- *rakia/event-target.[ch]*: helper to listen for events for a NUA handle (see NUA definition in sofia-sip documentation).
- *rakia/handles.[ch]*: helpers for *Handle*s.
- *rakia/sofia-decls.h*: some extra declaration
- *rakia/util.[ch]*: utility functions.

sofia-sip

sofia-sip[11] is a User-Agent library that implements the SIP protocol as described in IETF RFC 3261. It can be used for VoIP, IM, and many other real-time and person-to-person communication services. telepathy-rakia makes use of sofia-sip to implement SIP support into telepathy. sofia-sip has good documentation[12] on all concepts, events and APIs.

Connection Manager and creating connections

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

[11] http://sofia-sip.sourceforge.net/

[12] http://sofia-sip.sourceforge.net/refdocs/nua/

Once the Connection Manager Object construction is finalized, the creation of a SIP Protocol Object is triggered inside *rakia_connection_manager_constructed()* by calling *rakia_protocol_new()*. This function is defined in *src/protocol.c*. It creates a Protocol Object and adds the necessary infrastructure that a Connection Manager needs to manage the Protocol. In the Class Factory it is possible to see which methods are defined by this Class by looking at the *TpBaseProtocolClass base_class* var:

```
base_class->get_parameters = get_parameters;
base_class->new_connection = new_connection;
base_class->normalize_contact = normalize_contact;
base_class->identify_account = identify_account;
base_class->get_interfaces = get_interfaces;
base_class->get_connection_details = get_connection_details;
base_class->dup_authentication_types = dup_authentication_types;
```

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

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

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

```
sip_class->create_handle = rakia_connection_create_nua_handle;
sip_class->add_auth_handler =
rakia_connection_add_auth_handler;
```

and for the *TpBaseConnectionClass*:

```
base_class->create_handle_repos = rakia_create_handle_repos;
base_class->get_unique_connection_name = rakia_connection_unique_name;
base_class->create_channel_managers = rakia_connection_create_channel_managers;
base_class->create_channel_factories = NULL;
base_class->disconnected = rakia_connection_disconnected;
base_class->start_connecting = rakia_connection_start_connecting;
base_class->shut_down = rakia_connection_shut_down;
base_class->interfaces_always_present =
interfaces_always_present;
```

[13] http://telepathy.freedesktop.org/doc/telepathy-glib/TpBaseConnectionManager.html
[14] http://telepathy.freedesktop.org/doc/telepathy-glib/telepathy-glib-base-protocol.html

During the *TpBaseConnection* object construction the `create_channel_managers` method is called. A *Channel* is an entity provided by a *Connection* to allow the 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 *Channel*s can exist at a given time.

Channels and Calls

telepathy-rakia has two types of *Channel*s: *Text* and *Call*. For *TextChannel*s a *RakiaTextManager* objects is created. It inherits from *TpChannelManager*. *TpChannelManager* is a generic type used by all types of *Channel*s. 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 *Session*s and *Call Channel*s. 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)
{
    TpChannelManagerIface *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;
    iface->ensure_channel = rakia_media_manager_ensure_channel;
}
```

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 *Channel*s 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 it name says get called when a new call arrives.

*CallChannel*s, 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 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].

[15] http://telepathy.freedesktop.org/spec/Connection_Interface_Requests.html
[16] http://telepathy.freedesktop.org/spec/Channel_Type_Call.html#Property:InitialAudio
[17] http://telepathy.freedesktop.org/spec/Channel_Type_Call.html#Property:InitialVideo
[18] https://wiki.gnome.org/Folks
[19] http://telepathy.freedesktop.org/doc/folks/vala/Folks.Backend.html

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 it is extensive documentation. The EDS subclass for *Folks.Backend* is defined in *backend/eds/eds-backend.vala* in the sources.

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

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

The Telepathy backend also serves as a good example. As the EDS backend, it 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