Powered by Universal Speech Solutions LLC

Client Integration Manual

Developer Guide

Revision: 37

Last updated: May 20, 2017

Created by: Arsen Chaloyan

Table of Contents

1 Overview

This guide describes how to use the UniMRCP client library in a user application. The document provides basic steps only and is not a complete reference.
The intended audience is speech application developers familiar with the C/C++ programming languages.

1.1 Applicable Versions

Unless explicitly stated, instructions provided in this guide are applicable to the following versions.

UniMRCP 1.0.0 and above

1.2 Installation and Configuration

For the installation and configuration of the library, see the Installation Guide and the Client Configuration Guide.

2 Architecture

The UniMRCP client library provides a unified, MRCP version consistent interface for integration in IVRs, Call Centers, IP PBXs and other speech applications. The UniMRCP client stack is a multi-threaded library implemented in the C language and built on top of a number of internal and external components. The library uses the asynchronous event-driven model with a fixed number of threads (tasks) launched upon initialization. Memory pools are used throughout the library in order to avoid dynamic memory allocations and, thus, make memory operations highly efficient.

3 Client Stack Initialization

3.1 Creating an Instance of the Client Stack

Typically, one instance of the client stack is created per running process. The client stack is loaded from the configuration file(s) by default, although it is also possible to create and initialize the stack by means of API calls (not covered in this document). The location of the configuration directory is specified by the structure apt_dir_layout_t.

```
apt_dir_layout_t *dir_layout = apt_default_dir_layout_create(root_dir_path,pool);
mrcp_client_t *mrcp_client = unimrcp_client_create(dir_layout);
```

3.2 Creating an Instance of the Application Object

One or more instances of mrcp_application_t objects can be created and further registered to the client stack. An instance of mrcp_application_t is a logical representation of a user application communicating with the client stack. A message handler (mrcp_app_message_handler_f) must be provided upon creation of the application instance. The handler is used to receive messages sent from the client stack.

```
mrcp_application_t *mrcp_app = mrcp_application_create(app_message_handler,engine,pool);
mrcp_client_application_register(mrcp_client,mrcp_ap,name;
```

3.3 Starting the Message Processing Loop

The client stack is running in a loop waiting for messages either from the registered user applications and/or from other internal tasks.

- Synchronous start

  By default, the client stack is started synchronously. This means that the executable context of the calling user application will be blocked until the client stack is initialized and ready to process messages.

```
mrcp_client_start(mrcp_client);
```

- Asynchronous start

In order to start the client stack asynchronously, an event handler of type mrcp_client_handler_f must be set before attempting to start the message loop. This will allow to release the executable context of the calling user application while the stack is being initialized. The registered handler will be invoked asynchronously once the client stack is ready to process messages. Note that no requests can be sent from the application until the handler is invoked.

```
mrcp_client_async_start_set(mrcp_client,handler);
mrcp_client_start(mrcp_client);
```

4 MRCP Session Initialization

4.1 Creating an Instance of the MRCP Session

```
mrcp_session_t *mrcp_session = mrcp_application_session_create(mrcp_app,profile,NULL);
```

4.2 Creating an Instance of the MRCP Resource Channel

```
mrcp_channle_t *mrcp_channel = mrcp_application_channel_create(
              mrcp_session, MRCP_RECOGNIZER_RESOURCE, termination, NULL, obj);
```

4.3 Adding the Resource Channel to the Session

```
mrcp_application_channel_add(mrcp_session,mrcp_channel);
```

As a result, the client stack sends an offer to the MRCP server requesting to add/enable a resource channel. The request is processed asynchronously. The user application receives a response when the corresponding on_channel_add() callback is invoked by the client stack.

5 MRCP Message Management

Read the MRCP Message Usage guide in order to see how to create an MRCP message object, set its header fields and the body.

5.1 Sending an MRCP Request

```
mrcp_application_message_send(mrcp_session,mrcp_channel,mrcp_message);
```

This function sends a request to the MRCP server. A response is received asynchronously.

5.2 Receiving an MRCP Response or Event

```
on_message_receive(mrcp_app,mrcp_session_mrcp_channel,mrcp_message);
```

This callback is invoked when a response or event is received from the MRCP server.

6 MRCP Session De-initialization

6.1 Removing an MRCP Resource Channel

The allocated MRCP resource channel can optionally be de-allocated first. If not, the channel will be de-allocated with the session termination, anyway.

```
mrcp_application_channel_remove(mrcp_session,mrcp_channel);
```

This function sends an offer to the MRCP server requesting to remove/disable a resource channel. The request is processed asynchronously. The user application receives a response when the corresponding on_channel_remove() callback is invoked by the client stack.

6.2 Terminating an MRCP Session

```
mrcp_application_session_terminate(mrcp_session);
```

This function initiates session termination by sending corresponding request to the MRCP server. The user application receives a response when on_session_terminate() callback is invoked. Note that session termination can be initiated at any time during the lifetime of the session.

6.3 Destroying an MRCP Session

```
mrcp_application_session_destroy(mrcp_session);
```

This function ultimately destroys the MRCP session by releasing all the memory allocated in the scope of the session. Note that the session MUST be terminated first and can be destroyed only when the corresponding session termination response has been received.

7 Client Stack De-initialization

7.1 Shutting down Client Stack

```
mrcp_client_shutdown(mrcp_client);
```

This function stops the message processing loop and also terminates all other activities in the client stack.

7.2 Destroying Client Stack

```
mrcp_client_destroy(mrcp_client);
```

This function ultimately destroy the client stack and releases all the allocated memory.

8 Frequently Asked Questions

What is the unimrcpclient application?

The unimrcpclient is a sample user client application written in the C language based on the libunimrcpclient library.

What is the umc application?

The umc is another sample user client application written in the C++ language based on the libunimrcpclient library.

Should I use one of the frameworks introduces in sample applications?

You may use one of the sample applications as a base or a prototype for your own application. However, generally speaking, there is no such a requirement.

Should I create a new instance of the client stack for a new MRCP session?

No, you should not. One instance of the client stack is designed and also supposed to serve concurrent MRCP sessions.

What is mrcp_application_t and can multiple instances of mrcp_application_t be registered to the same client stack?

At least one instance of mrcp_application_t must be created and registered to the client stack. In this term, mrcp_application_t is a logical representation of the user client application. And yes, it is possible to have more than one instance of mrcp_application_t registered to the same client stack.

What is the message dispatcher interface (mrcp_app_message_dispatcher_t) and why is it exposed to the client user application interface?

The user client application MUST register a message handler in order to receive messages from the client stack. All the responses and events from the client stack are delivered asynchronously as mrcp_app_message_t objects, which must be further processed by the client user application. Depending on internal architecture of the application, the messages can be processed in the context of the client stack or be passed to and processed from one of internal threads of the user application.

9 References

9.1 Implementation

- unimrcpclient – a sample user client application written in C
- umc - a sample user client application written in C++
- mod_unimrcp – an MRCP module for FreeSWITCH
- res-speech-unimrcp – an MRCP based implementation of the Asterisk Generic Speech Recognition API
- app-unimrcp – an MRCP speech application module for Asterisk

9.2 Documentation

- UML Design Concepts – hierarchy, activity and sequence diagrams of the client stack.
- API Reference – a documentation generated by Doxygen from the source code