**Spotify Connect Design documentation**

**Introduction**

Spotify Connect allows users to stream music to (as well as control) hardware/devices from the Spotify app, whether that be on mobile or via the desktop and web player.

The device being used to stream music must be on the same Wi-Fi connection, and if you’re using a mobile device it’ll act as a remote so you can control playback, volume or add other devices to stream to.

Music doesn’t stream from the smartphone to speaker, but instead direct from the Spotify services to the speaker itself, which means your smartphone doesn’t use up any power as it would do if you’re using Bluetooth.

All you need to do is open the app on mobile, desktop or web player, click on the speaker icon at the bottom of the screen and select the device (or devices) that you want to stream to. That means a multi-room setup is possible as you can stream to multiple devices in different rooms.

**The RPI-Oradio as spotify connect device**

We need to two packages:

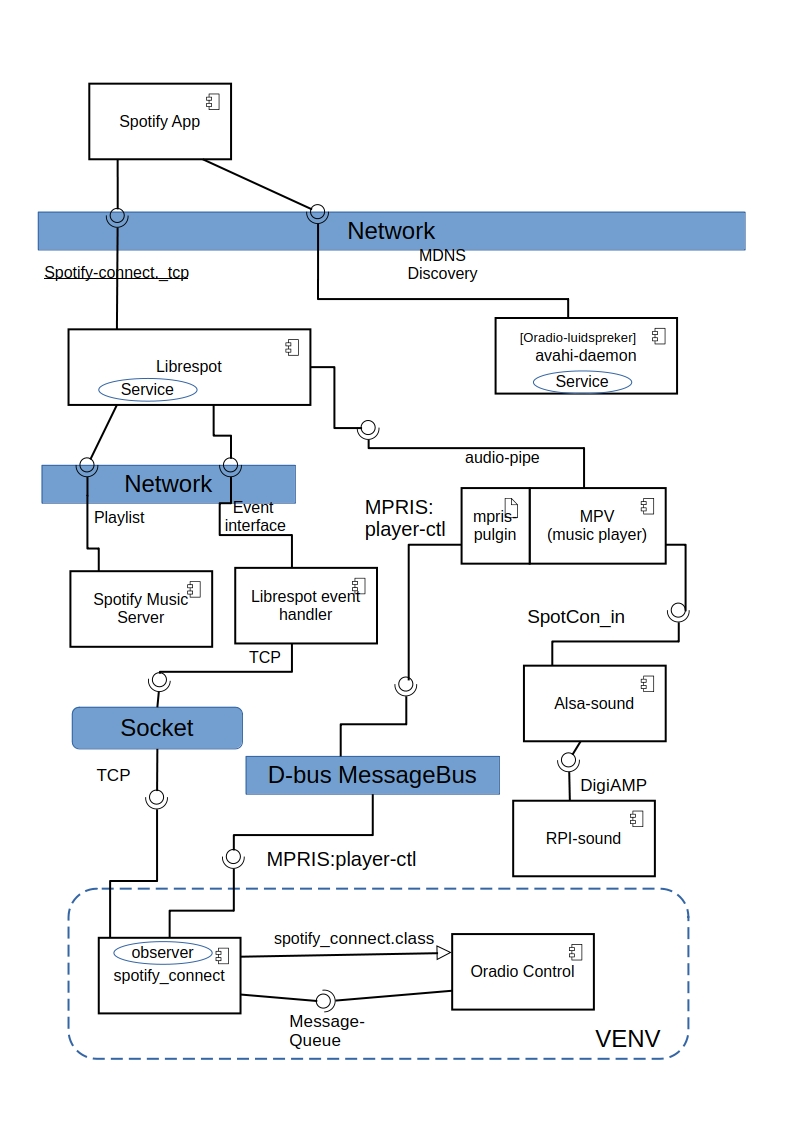
* The raspotify package: will turning our Raspberry Pi into a Spotify Connect device.
* The avahi-daemon package: is a free Zero-configuration networking (zeroconf) implementation, including a system for multicast DNS/DNS-SD service discovery.

Spotify uses mDNS to discover endpoints (hence a librespot instance will use mDNS to broadcast itself). By default, librespot uses its own mDNS responder implementation.

Avahi is a system daemon, that does multiple things:

* client: provide name resolution for mDNS services.
* responder: optionally, avahi will also broadcast services records for other programs (typically through dbus)
* It allows programs to publish and discover services and hosts running on a local network with no specific configuration. For example you can plug into a network and instantly find printers to print to, files to look at and people to talk to.

**Deployment (Component) view**

****

**Spotify App**

Searches for sound devices using the mDNS Discovery service of avahi-daemon. The Oradio advertises with <Oradio-luidspreker> as sound device.

When a Spotify Connect App (on a phone, tablet, or PC) searches for devices:

It sends a mDNS query for services of type <\_spotify-connect.\_tcp>. The avahi-daemon responds with the registered librespot service.

The Spotify App then lists the Oradio (librespot) as an available device.

Once the Spotify App selects the Oradio:

* It establishes a connection with Librespot using Spotify's authentication and control protocols.
* It sends commands like Play, Pause, Skip, Volume Control.
* Librespot then fetches and plays the audio directly from Spotify servers.

**Librespot Service**

After receiving a play command, Librespot:

* Authenticates with Spotify servers.
* Retrieves the requested audio stream.
* Decodes and outputs the audio to the Raspberry Pi’s sound card or external DAC.

If Librespot stops or restarts:

* It re-registers itself with avahi-daemon.
* If the Raspberry Pi’s IP address changes:
  + avahi-daemon updates the mDNS records dynamically.

**Librespot keep-alive mechanism**

The Pong messages in librespot are part of a keep-alive mechanism to maintain the connection with Spotify's servers. Here's what's happening:

1. Sending Pong → librespot is responding to a "Ping" from Spotify’s server.
2. ExpectingPongAck → librespot waits for an acknowledgment (PongAck) from Spotify, with a timeout of 20 seconds.
3. Received PongAck → The acknowledgment arrives, confirming the connection is still active.
4. ExpectingPing → librespot now waits for another "Ping" from Spotify, with a timeout of 80 seconds before it assumes the connection is dead.

### Why is this happening?

* This is a heartbeat mechanism used to detect connection drops and keep the session active.
* If librespot doesn’t receive a Ping or PongAck within the expected time, it may assume the connection is lost and attempt a reconnect.

**Audio-pipe**

Librespot provides an audio-pipe to the MPV player.

**Avahi-daemon:**

Librespot uses the avahi-api to publish the Spotify Connect Service.

This service contains metadata (TXT records) such as:

* Device name (e.g., “Oradio-luidspreker” )
* Supported sample rate, bit depth, and codec
* A unique device ID for Spotify authentication

**Librespot event handling:**

With librespot --onevent=/path/to/my/event/script/program users can subscribe to non-blocking events. The type of script/program you use to handle events is completely up to you, librespot simply runs the script/program every time an event is fired and passes events via environment variables. See for [events](https://github.com/librespot-org/librespot/wiki/Events)

### **librespot non-blocking events**

Non-blocking events are non-blocking in every sense. They do not block librespot threads in any way and the event script/program is run in it's own separate thread for each and every event. librespot's event handler does not wait for event scripts/programs to exit before it fires the next event.

#### Changed

| Key | Value | Description |
| --- | --- | --- |
| PLAYER\_EVENT | changed | Name of the Event |
| OLD\_TRACK\_ID | Spotify Track ID | ID of the previous Track |
| TRACK\_ID | Spotify Track ID | ID of the new Track |

#### Started

| Key | Value | Description |
| --- | --- | --- |
| PLAYER\_EVENT | started | Name of the Event |
| TRACK\_ID | Spotify Track ID | ID of the Track |

#### Stopped

| Key | Value | Description |
| --- | --- | --- |
| PLAYER\_EVENT | stopped | Name of the Event |
| TRACK\_ID | Spotify Track ID | ID of the Track |

#### Playing

| Key | Value | Description |
| --- | --- | --- |
| PLAYER\_EVENT | playing | Name of the Event |
| TRACK\_ID | Spotify Track ID | ID of the Track |
| DURATION\_MS | Milliseconds | Duration in ms |
| POSITION\_MS | Milliseconds | Position in ms |

#### Paused

| Key | Value | Description |
| --- | --- | --- |
| PLAYER\_EVENT | paused | Name of the Event |
| TRACK\_ID | Spotify Track ID | ID of the Track |
| DURATION\_MS | Milliseconds | Duration in ms |
| POSITION\_MS | Milliseconds | Position in ms |

#### Preloading

| Key | Value | Description |
| --- | --- | --- |
| PLAYER\_EVENT | preloading | Name of the Event |
| TRACK\_ID | Spotify Track ID | ID of the Track |

#### Volume Set

| Key | Value | Description |
| --- | --- | --- |
| PLAYER\_EVENT | volume\_set | Name of the Event |
| VOLUME | volume | Volume 0 - 65535 |

**Alsa-sound:**

Alsa-sound offers a sound device to be assigined to Librespot.

pcm.SpotCon\_in {

type plug

slave.pcm "SpotCon\_VolControl"

}

**Librespot playlist handling**

### The Playback State is Managed by Spotify's Cloud. When the playback is started from the Spotify App the Spotify servers:

* Store the current track and playlist/queue
* Maintain the playback progress (current timestamp in song).
* Track which device is currently active.
* Even if the Spotify app disconnects, librespot keeps playing because it receives streaming data directly from Spotify’s cloud.

Reconnecting to Spotify restores the playback.

If the Spotify app is closed and reopen it, the app queries Spotify’s servers to check:

* The last active device (librespot on the Raspberry Pi).
* The current song, playlist, and progress.
* The app then syncs back to the Oradio's playback state.
* This is why when you open Spotify, it might show:  
  "Playing on the Oradio-luidspreker", even if you left it playing earlier.

Librespot does not store playlist locally. When restarted librespot will reconnect to Spotify. If spotify detects that librespot was the last device it may resume playback where it left off.

**MPV player:**

mpv is a free and open-source media player software based on Mplayer, mplayer2 and Ffmpeg.

The mpv-mpris is a plugin for mpv which allows control of the player using the MPRIS D-Bus interface.

**MPRIS:**

MPRIS(Media Player Remote Interfacing Specification) is a standard D-Bus interface which aims to provide a common programmatic API for controlling media players.

MPRIS provides following methods:

* next
* previous
* pause
* playpause
* stop
* play
* seek
* setposition

**Spotify\_connect**

A spotify\_connect class to provide methods to control the MPV player via MPRID-player-ctl. Following method will be provided:

* start\_playing
* stop\_playing
* pause\_playing

Spotify\_connect has an observer to monitor new events coming from the spotify\_event\_handler socket. The events are listed in above section. There are 3 events which will be put in the message-queue for the statehandler in oradio\_control:

* play: can be used to start the mpv player using the method provided by the class
* pause: can be used to pause the mpv player using the method provided by the class
* stop:can be used to stop the mpv player using the method provided by the class

# Module Interface specification:

## Constants

MESSAGE\_SPOTIFY\_TYPE = *"Spotify message"*

*# SPOTIFY events and states*

SPOTIFY\_CONNECT\_PLAYING\_EVENT = *"Spotify Connect playing event"*

SPOTIFY\_CONNECT\_PAUSED\_EVENT = *"Spotify Connect paused event"*

SPOTIFY\_CONNECT\_STOPPED\_EVENT = *"Spotify Connect stopped event"*

SPOTIFY\_CONNECT\_CONNECTED\_EVENT = *"Spotify Connect connected event"*

SPOTIFY\_CONNECT\_DISCONNECTED\_EVENT = *"Spotify Connect disconnected event"*

SPOTIFY\_CONNECT\_CLIENT\_CHANGED\_EVENT = *"Spotify Connect client changed event"*

*# MPV\_PLAYER COMMANDS ####*

MPV\_PLAYERCTL\_PLAY = *"play"*

MPV\_PLAYERCTL\_PAUSE = *"pause"*

MPV\_PLAYERCTL\_STOP = *"stop"*

*#MPV\_PLAYER STATES ####*

MPV\_PLAYERCTL\_PLAYING\_STATE = *"playing"*

MPV\_PLAYERCTL\_STOPPED\_STATE = *"stopped"*

MPV\_PLAYERCTL\_PAUSED\_STATE = *"paused"*

*SPOTIFY\_CONNECT\_SERVERS\_RUNNING* = *"Spotify Connect local serversunning"*

SPOTIFY\_CONNECT\_SERVERS\_NOT\_RUNNING = *"Spotify Connect local servers NOT running"*

SPOTIFY\_CONNECT\_MPV\_SERVICE\_NOT\_ACTIVE = *"Spotify Connect MPV service not active"*

SPOTIFY\_CONNECT\_MPV\_SERVICE\_IS\_ACTIVE = *"Spotify Connect MPV service is active"*

SPOTIFY\_CONNECT\_MPV\_MPRIS\_PLAYER\_NOT\_FOUND = *"Spotify Connect MPV MPRIS player not found"*

SPOTIFY\_CONNECT\_MPV\_STATE\_OK = *"Spotify Connect MPV State OK"*

## Message-queue events

Uses the Message schema as defined in schemas.json

{

"$schema": "http://json-schema.org/draft-07/schema#",

"Messages": {

"type": "object",

"properties": {

"type": { "type": "string" },

"state": { "type": "string" },

"error": { "type": "string" },

"data": { "type": "array" }

},

"required": ["type", "state", "error"]

}

message[*"type"*] = MESSAGE\_SPOTIFY\_TYPE

message[*"state"*] = SPOTIFY\_CONNECT\_PLAYING\_EVENT

message[*"state"*] = SPOTIFY\_CONNECT\_PAUSED\_EVENT

message[*"state"*] = SPOTIFY\_CONNECT\_STOPPED\_EVENT

message[*"state"*] = SPOTIFY\_CONNECT\_CONNECTED\_EVENT

message[*"state"*] = SPOTIFY\_CONNECT\_DISCONNECTED\_EVENT

message[*"state"*] = SPOTIFY\_CONNECT\_CLIENT\_CHANGED\_EVENT

For all message[*"state"*] the message[*"error"*] = *"None"*

## SpotifyConnect class methods:

class **SpotifyConnect**():

def **\_\_init\_\_**(*self*, msg\_queue):

def **get\_state**(*self*):

*'''*

*Return the actual state of the Spotify Connect servers and playback status*

*:return spotify\_app\_status = [ SPOTIFY\_APP\_STATUS\_PLAYING | SPOTIFY\_APP\_STATUS\_STOPPED |*

*SPOTIFY\_APP\_STATUS\_PAUSED | SPOTIFY\_APP\_STATUS\_DISCONNECTED |*

*SPOTIFY\_APP\_STATUS\_CONNECTED | SPOTIFY\_APP\_STATUS\_CLIENT\_CHANGED]*

*:return spotify\_connected\_state = [ SPOTIFY\_CONNECT\_CONNECTED | SPOTIFY\_CONNECT\_NOT\_CONNECTED]*

*'''*

def **playerctl\_command**(*self*,command):

*'''*

*Send command to playerctl via mpris player interface*

**:param** *command = player command = [ MPV\_PLAYERCTL\_PLAY,*

*MPV\_PLAYERCTL\_PAUSE,*

*MPV\_PLAYERCTL\_STOP]*

**:return** *self.state = state of mpv mpris player-control = [ MPV\_PLAYERCTL\_PLAYING\_STATE | MPV\_PLAYERCTL\_STOPPED\_STATE |*

*MPV\_PLAYERCTL\_PAUSED\_STATE]*

*'''*

## Activity view diagrams

### Activity view for play-pause at the Spotify App



### Activity view for play-disconnect at Spotify App and local play-pause

