Creating a New Endpoint: Lifecycle of a Session

0. About

FreeSWITCH endpoints are designed to create, handle and destroy media streams. This page describes sessions and channels from the point of view of writing an extension module for FreeSWITCH.

1. Endpoints

FreeSWITCH manages the establishment and destruction of linked sessions between entities. For those of us from a telephony background, FreeSWITCH performs the functions of a PBX or central office. The following document is written with respect to voice calls, but the principles apply to the other types of connections that FreeSWITCH can manage.

The voice entities that we’re most familiar with are simple telephones. The modules that handles these are called ‘endpoints’. The most used endpoint modules are mod_sofia (SIP phones) and mod_skinny (Cisco phones).

As modules, they must have the following entry points to be loaded and unloaded:

- module_load()
- module_unload()

TODO Couldn’t find direct references to module_load() and module_unload() in Creating New Modules, so check what this section is referring to.

Usually a module will also implement two interfaces, as arrays of callback functions. This is how FreeSWITCH communicates with the module.

An endpoint module deals a lot with sessions.

2. Sessions

A FreeSWITCH session is equivalent to a call-leg when the session is managed by mod_sofia.

A session contains a channel and private data.

This diagram shows containment, when in fact pointers are used.

A session object owns the channel, so in a sense the session is the parent of the channel. The session also wraps the channel in some management overhead, including:

- thread to run the code
- memory pool
- array of codecs
- message queues
- read/write data buffers
The *session* also owns a *private* object. The author of the module defines what this object contains and what it’s used for. In almost all modules, this object is called `session->tech_pvt`. Some common elements of a `tech_pvt` are:

- IP addresses involved in the communication channel
- encryption keys for the communication channel
- the external codecs involved in the media flow

![Diagram of session, channel, and tech_pvt objects]

### 3. Channels

The *channel* is the basic entity for one end of a FreeSWITCH call. *Channel* objects manage the flow of media to the outside world. *Channels* are where the most interesting things happen. The *channel* runs through *states* that represent the state of call setup, the media flow and the call tear down.

The most interesting members of a *channel* (see `switch_channel.c`) are:

- name
- uuid
- direction
- its *session* (wrapper)
- state
- caller_profile
- flags
- variables

#### 3.1 state

The single most important *channel* member is the *state* variable, designed to handle the connection status. The *state* member is the one that most FreeSWITCH documentation references. This state runs through the sequence `CS_NEW, CS_INIT, CS_ROUTING, CS_EXECUTE, CS_HANGUP, CS_REPORTING, CS_DESTROY`. The transitions from one state to the other are handled gracefully by the internals of FreeSWITCH (`switch_core_state_machine.c`) unless you intercept these transitions. This state describes the highest level connection status of the *channel/session*.

The *state* variable is not directly what one would normally use in telephony. Because FreeSWITCH can handle many kinds of media, the *state* variable is designed to handle the connection status. You need also look at `CF_RING_READY`, `CF_EARLY_MEDIA` and `CF_ANSWERED` in order to determine the telephony state.

Here’s a table to illustrate some details of the states:

<table>
<thead>
<tr>
<th>State</th>
<th>Normal activities</th>
<th>Standard processing</th>
<th>Your module code should….</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS_NEW</td>
<td>right after <em>session</em> creation</td>
<td>none</td>
<td>change to CS_INIT</td>
</tr>
</tbody>
</table>
CS_INIT  create any data structures required for the rest of the session/call
CS_ROUTING  inbound: dial tone, accepting digits, ringback
            outbound: uses the caller_profile & the dialplan to pick a destination
           inbound: your module will get a message SWITCH_INDICATE_ANSWER
            outbound: your module should call switch_call_answer()
CS_CONSUME_MEDIA  outbound: ringing
CS_EXCHANGE_MEDIA  outbound: talking
CS_EXECUTE  inbound: media flows; talking
            outbound: near end release: your module should call switch_channel_hangup()
            far end release: your module will get a callback on my_module_on_hangup()
CS_HANGUP  Changes to CS_REPORTING*
CS_REPORTING  Changes to CS_DESTROY*
CS_DESTROY  Release any memory

3.2 running_state and callstate

The channel/object also has two other state members (running_state and callstate) which are hardly used at all in the core of FreeSWITCH, and which you probably won’t need. The running_state is just used in state transitions, usually holds the "previous state". The callstate is a more fine-grained version of the state, and it has its own getter and setter. If you chose to use the setter, it will fire an event. callstate includes CCS_DOWN, CCS_DIALING, CCS_RINGING, CCS_EARLY, CCS_ACTIVE and CCS_HANGUP.

3.3 caller_profile

Within the channel is a single caller_profile. This contains descriptions of both the caller and callee.

The internal sections of FreeSWITCH only use:

- uuid
- caller_id_name, caller_id_number, caller_extension
- dialplan
- destination_number

The channel/flags hold extra state information above and beyond the basic 13 states that are held in the state variable. The interesting flags are:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CF_ANSWERED</td>
<td>Channel is answered</td>
</tr>
<tr>
<td>CF_OUTBOUND</td>
<td>Channel is an outbound channel</td>
</tr>
<tr>
<td>CF_EARLY_MEDIA</td>
<td>Channel is ready for audio before answer</td>
</tr>
<tr>
<td>CF_ORIGINATOR</td>
<td>Channel is an originator</td>
</tr>
<tr>
<td>CFTRANSFER</td>
<td>Channel is being transferred</td>
</tr>
<tr>
<td>CF_HOLD</td>
<td>Channel is on hold</td>
</tr>
<tr>
<td>CF_GEN_RINGBACK</td>
<td>Channel is generating its own ringback</td>
</tr>
<tr>
<td>CF_RING_READY</td>
<td>Channel is ready to send ringback</td>
</tr>
<tr>
<td>CF_BREAK</td>
<td>Channel should stop what it’s doing</td>
</tr>
</tbody>
</table>

3.4 Variables

A channel also has a hash of key-value pairs that can hold all kinds of interesting data related to the channel. These are used extensively by mod_sofia to bind things like sip_user_agent and sip_profile_name to a channel. These key-value pairs can be queried by either the dialplan or LUA processing. They can also be a handy way to accumulate data in preparation for issuing a FreeSWITCH event. See channel variables.

4. Howto Write a FreeSWITCH Endpoint

First follow the New Module Checklist. Earlier directions suggest using freeswitch/src/mod/applications/mod_skel to start off, but for endpoints you’re better off starting with freeswitch/src/mod/endpoints/mod_reference.
Module Load is the first entry point called. Usually at this point you will inform the caller about your module, its callback interface arrays and register any API extensions. You may also want to create a memory pool. You should read your config data, which by convention is XML, and buried somewhere in freemeswitch/conf.

5. Handling calls from within your module

Lifecycle of a FreeSWITCH call

Below is our current understanding. We are not core developers, so all is guessed –Sathieu 11:14, 18 February 2010 (UTC) – PatB 2015-09-25

5.1 Common tasks to both incoming and outgoing calls

For both outgoing and incoming calls, you must allocate the session. The core of FreeSWITCH will automatically allocate and bind a channel to your session. FreeSWITCH does not create the caller_profile; your code must allocate a caller_profile and bind it to the channel. You probably also want to allocate a private data object, and bind it to the session.

Notice that in a normal call, the inbound/originating channel ends up in the CS_EXECUTE state, and the outbound/called channel ends up in the CS_EXCHANGE_MEDIA state. [Once we start playing with 'hold' and 'transfer' this statement gets muddied.]

If your module wants to shut down the call, just use switch_channel_hangup(). You will get a callback to your on_hangup() entry point and the call will shut down.

If the other end wants to shut down the call, you will get a SIG_KILL sent to your on_channel_kill() entry point. Shortly thereafter you will get a call to your on_hangup() entry point and the call will shut down.

Notes

[1] you don't need to create the channel...it is created inside the session
[2] at minimum, you should set the .name member of the channel
[3] or if you wish to abort, call switch_channel_hangup(channel)
[4] strangely, you only call s_c_s_thread_launch() on incoming calls; it's called automatically for you on outgoing calls (deep in switch_ivr.c)

5.2 Outgoing calls

Outgoing calls are created either by a call coming through the dialplan routing, or when you use the originate command in the console:

* your_io_routines.outgoing_channel() is called
  * you get a pointer to the other call half (session/channel)
  * create a new session with switch_core_session_request() and private data[1]
  * parse outbound_profile->destination_number to get called number
  * create a caller_profile and bind it to the session
  * qualify the session, attached channel and attached private data[2]
  * the channel is currently in CS_NEW; change channel state to CS_INIT
    * make the phone ring (and call switch_channel_ring_ready())
  * your_state_handler.on_init() is called
    * no need to do anything
    * the default handler will change channel/state to CS_ROUTING
  * your_state_handler.on_routing() is called
    * no need to do anything
    * the default handler will change channel/state to CS_CONSUME_MEDIA
    * when you decide the call is answered
      * call switch_channel_mark_answered(channel) [3] which will set the channel/state to CS_EXCHANGE_MEDIA
  * your_state_handler.on_exchange_media() is called
    * and the call is up

5.3 Incoming calls

Incoming calls are initiated by your code. To start up a call
• create a new session (with switch_core_session_request()) and private data[1]
  • create a caller_profile (with switch_caller_profile_new()) and fill in the ->destination_number
  • qualify the channel and private data [2]
  • create a caller_profile and bind it to the channel
  • start up the thread that manages the session (with switch_core_session_thread_launch()) [4]
  • change channel state from CS_NEW to CS_INIT
• your_state_handler.on_init() will be called
  • no need to do anything
  • the default handler will change channel state to CS_ROUTING
• your_state_handler.on_routing() is called
  • no need to do anything
  • if your caller_profile->destination_number has been set, the default handler will search the dialplan and start an application; if this is a telephone call, usually the ‘bridge’ application will start an outbound session toward the called party and change change this channel state to CS_EXECUTE
• when the distant party answers
  • your_state_handler.on_receive_message() is called
    • filter the messages you get for SWITCH_MESSAGE_INDICATE_ANSWER
    • take any action that you need to start up the call

From the directory that you installed freeswitch, you can simply type "make mod_my_module" to just remake your loadable module. In the FreeSWITCH console, you can enter load mod_my_module and unload mod_my_module as often as you want, fixing bugs between builds.

6. The Babel tower of the API

When trying to learn FreeSWITCH programming, you’re going to be overwhelmed by the number of API methods that are available. Here’s a list on the one’s you’re likely to need in session and channel management.

<table>
<thead>
<tr>
<th>API Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>switch_core_session_request()</td>
<td>alloc and initialize a new session</td>
</tr>
<tr>
<td>switch_core_session_get_channel()</td>
<td>given a session, get the associated channel</td>
</tr>
<tr>
<td>switch_core_session_get_private()</td>
<td>given a session, get the private data</td>
</tr>
<tr>
<td>switch_channel_get/set_state()</td>
<td>used to coerce a channel state; usually the only one you will do is CS_NEW/CS_INIT</td>
</tr>
<tr>
<td>switch_core_session_alloc()</td>
<td>alloc space, usually for tech_pvt</td>
</tr>
<tr>
<td>switch_caller_profile_clone()</td>
<td>on outgoing, clone the caller’s caller_profile</td>
</tr>
<tr>
<td>switch_channel_get/set_caller_profile()</td>
<td>bind a caller_profile to a channel</td>
</tr>
<tr>
<td>switch_channel_test_flag()</td>
<td>set/test CF_ANSWERED, CF_EARLY_MEDIA, etc</td>
</tr>
<tr>
<td>switch_channel_set_flag_value()</td>
<td></td>
</tr>
<tr>
<td>switch_channel_get/set_name()</td>
<td>give your channel a name, like “superphone/1001”</td>
</tr>
<tr>
<td>switch_channel_ring_ready()</td>
<td>ignored if the channel is already answered</td>
</tr>
<tr>
<td>switch_channel_mark_ring_ready()</td>
<td>if inbound, sends the message SWITCH_MESSAGE_INDICATE_RINGING</td>
</tr>
<tr>
<td>switch_channel_ring_ready()</td>
<td>calls switch_channel_mark_ring_ready()</td>
</tr>
<tr>
<td>switch_channel_mark_ring_ready()</td>
<td>set CF_RING_READY</td>
</tr>
<tr>
<td></td>
<td>sends a PROGRESS event</td>
</tr>
<tr>
<td></td>
<td>sets callstate=CCS_RINGING</td>
</tr>
<tr>
<td></td>
<td>sends message SWITCH_MESSAGE_RING_EVENT</td>
</tr>
<tr>
<td>Function Name</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>switch_channel_pre_answer()</td>
<td>Ignored if the channel is already answered. If inbound, sends the message SWITCH_MESSAGE_INDICATE_PROGRESS. Calls switch_channel_mark_pre_answer().</td>
</tr>
<tr>
<td>switch_channel_mark_pre_answer()</td>
<td>Sets CF_EARLY_MEDIA. Sends a PROGRESS_MEDIA event. Sets callstate=CCS_EARLY. Sends message SWITCH_MESSAGE_PROGRESS_EVENT.</td>
</tr>
<tr>
<td>switch_channel_answer()</td>
<td>Ignores outbound calls or calls already answered. Sends message SWITCH_MESSAGE_INDICATE_ANSWER. Calls switch_channel_mark_answered().</td>
</tr>
<tr>
<td>switch_channel_mark_answered()</td>
<td>Ignores calls already answered. Sets CF_ANSWERED. Sends CHANNEL_ANSWER event. Sets callstate=CCS_ACTIVE. Sends message SWITCH_MESSAGE_ANSWER_EVENT.</td>
</tr>
<tr>
<td>switch_channel_hangup()</td>
<td>Changes channel state to CS_HANGUP. Sends the HANGUP event. Sends SIG_KILL(1) to itself.</td>
</tr>
<tr>
<td>switch_core_session_thread_launch()</td>
<td>Starts up the state machine; inbound only.</td>
</tr>
<tr>
<td>switch_channel_direction()</td>
<td>Checks inbound or outbound.</td>
</tr>
</tbody>
</table>

There are several similar calls with the word "_perform_" in the name; these are just versions that include the filename/linenumber for debugging. Don't call those directly.

There are three pairs in the middle, with and without the _mark_ in the name. You call

```
switch_channel_ring_ready()
switch_channel_pre_answer()
switch_channel_answer()
```

when you want to issue the SWITCH_MESSAGE_INDICATE_"_" messages into the message queue. If you're responding to an INDICATE message, you don't need to re-issue that message, so you would call these below:

```
switch_channel_mark_ring_ready()
switch_channel_mark_pre_answer()
switch_channel_mark_answered()
```

For example, if you just got a SWITCH_MESSAGE_INDICATE_ANSWER, you would only call the switch_channel_mark_answered() version. If you are working on something else, and you decide the call should be answered, call the non-_mark_ version switch_channel_answer().
In practice, the standard state code does a pretty good job of managing channel flags, channel variable and channel states. The only routines that you will probably need to call are

```
switch_channel_set_state() // once, during session creation
switch_channel_mark_answered() // when your device answers
switch_channel_hangup() // to shut down
```

**Source code for those UML diagrams above:**

using PlantUML syntax

Fig 1

```
session --> channel
channel --> caller_profile
session --> tech_pvt
```

Fig 2

```
session --> channel
channel --> caller_profile
session --> tech_pvt

session: thread
session: memory pool
session: codec[]
session: messages[]

channel: name
channel: uuid
channel: direction
channel: flags[]
channel: variables[]

caller_profile: destination_number
caller_profile: caller_id

tech_pvt: ip addresses
tech_pvt: other data that you may need
```